Studies in Computational Intelligence 520

Mohammad Ayoub Khan
Saqib Saeed
Ashraf Darwish
Ajith Abraham *Editors*

Embedded and Real Time System Development: A Software Engineering Perspective

Concepts, Methods and Principles

Springer

Studies in Computational Intelligence

Volume 520

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

For further volumes:
http://www.springer.com/series/7092

About this Series

The series "Studies in Computational Intelligence" (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

Mohammad Ayoub Khan
Saqib Saeed · Ashraf Darwish
Ajith Abraham
Editors

Embedded and Real Time System Development: A Software Engineering Perspective

Concepts, Methods and Principles

Springer

Editors
Mohammad Ayoub Khan
Department of Computer Science and Engineering
School of Engineering and Technology
Sharda University
Greater Noida
India

Saqib Saeed
Department of Computer Sciences
Bahria University
Islamabad
Pakistan

Ashraf Darwish
Faculty of Science
Helwan University
Cairo
Egypt

Ajith Abraham
Machine Intelligence Research Labs
Scientific Network for Innovation and Research Excellence
Auburn
USA

ISSN 1860-949X ISSN 1860-9503 (electronic)
ISBN 978-3-642-40887-8 ISBN 978-3-642-40888-5 (eBook)
DOI 10.1007/978-3-642-40888-5
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013953267

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The software is the driving force for today's smart and intelligent products. The products and services have become more instrumented and intelligent. This trend of interconnection between product and services is stretching software development organizations and traditional software development model approaches to the limit. Now a day's embedded and real-time system contains complex software. The complexity of embedded systems is increasing, and the amount and variety of software in the embedded products are growing. This creates a big challenge for embedded and real-time software development process. To reduce the complexity of development cycle many development companies and researcher has paid their attention to optimize the timeliness, productivity, and quality of embedded software development and apply software engineering principles in embedded systems. Unfortunately, many available software development model do not take into account the specific needs of embedded and systems development. The software engineering principles for embedded system should address specific constraints such as hard timing constraints, limited memory and power use, predefined hardware platform technology, and hardware costs.

There is a need to develop separate metrics and benchmarks for embedded and real-time system. Thus, development of software engineering principles for embedded system and real-time system has been presented as an independent discipline.

The book presents practical as well as conceptual knowledge of the latest tools, techniques, and methodologies of embedded software engineering and real-time systems. Each chapter presents the reader with an in-depth investigation regarding the actual or potential role of software engineering tools in the context of the embedded system and real-time system. The book presents state of the art and future perspectives of embedded system and real-time system technologies, where industry experts, researchers, and academicians had shared ideas and experiences surrounding frontier technologies, breakthrough, and innovative solutions and applications.

Organization of the Book

The book is organized into four parts and altogether 12 chapters. Part I, titled "Embedded Software Development Process," and contains "A Flexible Framework for Component-Based Application with Real-Time Requirements and its Supporting Execution Framework" and "Automatic Development of Embedded Systems Using Model Driven Engineering and Compile-Time Virtualisation". Part II, named "Design Patterns and Development Methodology", contains "MADES EU FP7 Project: Model-Driven Methodology for Real Time Embedded Systems"–"A Fuzzy Cuckoo-Search Driven Methodology for Design Space Exploration of Distributed Multiprocessor Embedded Systems". Part III, titled "Modeling Framework", contains "Model-Based Verification and Validation of Safety-Critical Embedded Real-Time Systems: Formation and Tools"–"A Multi-Objective Framework for Characterization of Software Specifications". Part IV, titled "Performance Analysis, Power Management and Deployment", contains "An Efficient Cycle Accurate Performance Estimation Model for Hardware Software Co-Design"–"Software Deployment for Distributed Embedded Real-Time Systems of Automotive Applications". A brief description of each of the chapters follows:

"A Flexible Framework for Component-Based Application with Real-Time Requirements and its Supporting Execution Framework," presents fundamental process of embedded real-time systems to support component-based development process, and the schedulability analysis of the resulting software. Authors have proposed Model-Driven Software Engineering paradigm and its associated technologies. The development process of the proposed model is accompanied by an Eclipse-based tool-chain, and a sample case study.

"Automatic Development of Embedded Systems Using Model Driven Engineering and Compile-Time Virtualisation," presents discussion on application of model-driven engineering and compile-time virtualization. This chapter focuses upon new tools for the generation of software and hardware for modern embedded systems. The presented approach promotes rapid deployment and design space exploration. This integrated fully model-driven tool flow supports existing industrial practices. The presented approach also has provision for automatic deployment of architecture-neutral Java code over complex embedded architectures.

"MADES EU FP7 Project: Model-Driven Methodology for Real Time Embedded Systems," presents a complete methodology for the design of RTES in the scope of the EU funded FP7 MADES project. MADES aims to develop novel model-driven techniques to improve existing practices in development of RTES for avionics and surveillance embedded systems industries. It proposes an effective subset of existing standardized UML profiles for embedded systems modeling.

"Test-Driven Development as a Reliable Embedded Software Engineering Practice", presents Test-Driven Development (TDD) promotes testing software during its development, even before the target hardware becomes available.

Principally, TDD promotes a fast feedback cycle in which a test is written before the implementation. Authors have presented four different evaluation methods for TDD. Also, a number of relevant design patterns are discussed to apply TDD in an embedded environment.

"A Fuzzy Cuckoo-Search driven methodology for Design Space Exploration of Distributed Multiprocessor Embedded Systems", discusses a methodology for conducting a Design Space Exploration (DSE) for Distributed Multi-Processor Embedded systems (DMPE). Authors have used fuzzy rule-based requirements elicitation framework and Cuckoo-Search for the DMPE.

"Model-Based Verification and Validation of Safety-Critical Embedded Real-Time Systems: Formation and Tools," presents a new concept of Verification, Validation, and Testing (VV&T). The chapter covers software engineering to system engineering with VV&T procedures for every stage of system design, e.g., static testing, functional testing, Unit testing, fault injection testing, consistency techniques, Software-In-The-Loop (SIL) testing, evolutionary testing, Hardware-In-The-Loop (HIL) testing, Black box testing, White box testing, Integration testing, system testing, system integration testing, etc.

"A Multi-Objective Framework for Characterization of Software Specifications," presents the complexity of embedded systems. The complexity is exploding into two interrelated but independently growing directions: architecture complexity and application complexity. Authors have discussed a general purpose framework to satisfy multiple objectives of early design space exploration. Authors have proposed a multi-objective application characterization framework based on a visitor design pattern. Authors have used MPEG-2 video decoder as benchmark that shows viability of the proposed framework.

"An Efficient Cycle Accurate Performance Estimation Model for Hardware Software Co-Design," presents a proposal for performance estimation. Authors have measured the performance of software in terms of clock cycles. In this measurement the availability of hardware platform is critical in early stages of the design flow. The authors have proposed to implement the hardware components at cycle-accurate level such that the performance estimation is given by the micro-architectural simulation in number of cycles. Authors have measured the performance as a linear combination of function performances on mapped components. The proposed approach decreases the overall simulation time while maintaining the accuracy in terms of clock cycles.

"Multicast Algorithm for 2D de Bruijn NoCs," presents De Bruijn topology for future generations of multiprocessing systems. Authors have proposed de Bruijn for Networks-on-Chips (NoCs). Also, the chapter proposes a multicast routing algorithm for two-dimensional de Bruijn NoCs. The proposed routing compared with unicast routing using Xmulator simulator under various traffics.

"Functional and Operational Solutions for Safety Reconfigurable Embedded Control Systems," presents run-time automatic reconfigurations of distributed embedded control systems following component-based approaches. Authors have proposed solutions to implement the whole agent-based architecture, by defining UML meta-models for agents. Also, to guarantee safety reconfigurations of tasks

at run-time, a service and reconfiguration processes for tasks, and use the semaphore concept to ensure safety mutual exclusions is defined.

"Low Power Techniques for Embedded FPGA Processors," presents low power techniques for embedded FPGA processors. Authors have emphasized that clock signals is a great source of power dissipation because of high frequency and load. Authors have presented investigation and simulation of clock gating technique to disable the clock signal in inactive portions of the circuit. The chapter also presents Register-Transfer Level model in Verilog language.

"Software Deployment for Distributed Embedded Real-Time Systems of Automotive Applications," presents a deployment model for automatic applications. The chapter discussed the software deployment problem, tailored to the needs of the automotive domain. Thereby, the focus is on two issues: the configuration of the communication infrastructure and how to handle design constraints. It is shown, how state-of-the-art approaches have to be extended in order to tackle these issues, and how the overall process can be performed efficiently, by utilizing search methodologies.

Who and How to Read this Book

This book has three groups of people as its potential audience, (i) undergraduate students and postgraduate students conducting research in the areas of embedded software engineering and real-time systems; (ii) researchers at universities and other institutions working in these fields; and (iii) practitioners in the R&D departments of embedded systems. This book differs from other books that have comprehensive case study and real data from software engineering practices.

The book can be used as an advanced reference for a course taught at the postgraduate level in embedded software engineering and real-time systems.

Mohammad Ayoub Khan
Saqib Saeed
Ashraf Darwish
Ajith Abraham

Contents

Part I
Embedded Software Development Process

A Flexible Framework for Component-Based Application with Real-Time Requirements and its Supporting Execution Framework

Diego Alonso, Francisco Sánchez-Ledesma, Juan Pastor and Bárbara Álvarez

Abstract This chapter describes a development approach for supporting a component-based development process of real-time applications, and the schedulability analysis of the resulting software. The approach revolves around the Model-Driven Software Engineering paradigm and its associated technologies. They provide the theoretical and technological support for defining the most suitable abstraction levels at which applications are designed, analyzed, deployed, etc., as well as the automatic evolution of models through the defined abstractions levels. To ensure that the analyzed models correspond to the input architectural description, it is necessary to establish univocal correspondences between the concepts of the domains involved in the process. The development process is supported by an Eclipse-based tool-chain, and a sample case study comprising the well-known *cruise control problem* illustrates its use.

1 Introduction and Motivation

Developing software for *Real-Time* (RT) systems is a challenging task for software engineers. Since these systems have to interact with both the environment and human operators, they are subject to operational deadlines. Also, it is essential that they be so designed as to involve no risk to the operators, the environment, or the system itself. Thus, designers have to face two different problems, namely **software design** and **software analysis**, complicated by the fact that time plays a central role in RT systems. There are many well-known software disciplines that provide solutions to each of the aforementioned problems in the literature:

D. Alonso (✉) · F. Sánchez-Ledesma · J. Pastor · B. Álvarez
Department of TIC, Universidad Politécnica de Cartagena, Cuartel de Antigones, 30202 Cartagena, Spain
e-mail: diego.alonso@upct.es

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_1, © Springer-Verlag Berlin Heidelberg 2014

Software design: Software Architecture constitutes the backbone for any successful software-intensive system, since it is the primary carrier of a software system's quality attributes [27]. Component-Based Software Development (CBSD) is a bottom-up approach to software development, in which applications are built from small modular and interchangeable unit, with which the architecture of a system can be designed and analyzed [29]. Frameworks [17] and design patterns [18] are the most successful approaches to maximize software quality and reuse available nowadays.

Software analysis: Software analysis is, perhaps, a broader area than software design, since there are many characteristics that can be analyzed in a piece of software, depending on the needs of each stakeholder. Thus, it is possible to use model checking [4], validation and verification tools [5, 6], schedulability analysis tools [21, 28], to mention but just a few.

However, is it very difficult to combine the results from both disciplines, since it implies to reconcile the design and analysis worlds, which are concerned with very different application aspects, and therefore use very different concepts: components the former and threads the latter. To ensure that the analyzed models correspond to the input architectural description, it is necessary to establish univocal correspondences between the concepts of both domains. There are different ways of defining such correspondences, but most of them imply constraining the implementation to just a few alternatives, when it would be desirable to select among various alternatives. Typical examples of this are component models that implement components as processes; or those where all components are passive and invoked sequentially by the run-time; or those that enforce a given architectural style, like pipes & filters, etc.

This chapter describes a flexible development approach for supporting a component-based development process of real-time applications, and the schedulability analysis of the resulting software. The word "flexible" in the previous sentence is used to emphasize that our work does not impose a rigid implementation but rather provides the user with some implementation options, as described in the rest of the chapter. The approach revolves around the *Model-Driven Software Development* (MDSD) paradigm [12, 26] and its associated technologies. They provide the theoretical and technological support for defining the most suitable abstraction levels at which applications are designed, analyzed, deployed, etc., as well as the automatic evolution of models through the defined abstractions levels. Thanks to model transformations, models can automatically evolve from design to analysis without the user having to make such transformation manually.

The approach described in this chapter comprises three abstraction levels, namely: (1) architectural software components for designing applications, (2) processes for configuring the application deployment and concurrency, and (3) threads and synchronization primitives for analyzing its schedulability. Figure 1 represents the relationships existing among these levels by using the well-known MDSD pyramids. This process has been integrated in an Eclipse-based tool-chain, also described in this chapter.

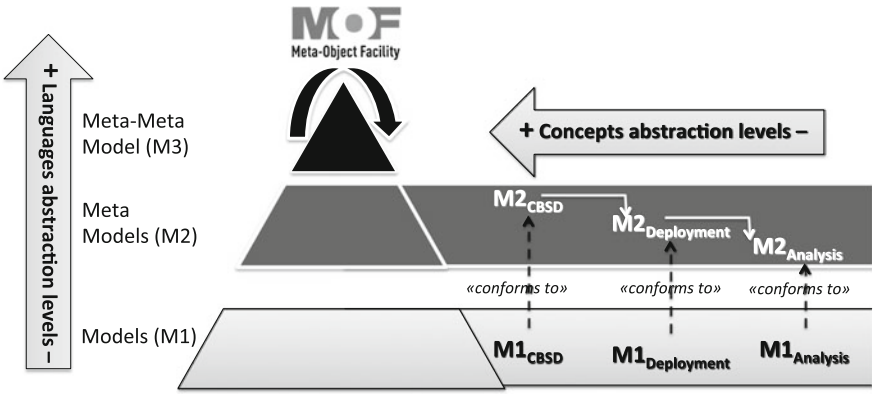


Fig. 1 Considered abstraction levels, organized in the well-known MDSD pyramid from two orthogonal points of view: modeling languages and concepts

MDSD is an emerging paradigm aimed at raising the level of abstraction during the software development process. MDSD uses models as first-class artifacts, allowing designers to create models with the desired/required level of detail. Two central key concepts to MDSD are models and model transformations (which are themselves considered models). Models represent part of the functionality, structure and/or behavior of a system. They are defined in terms of formal meta-models, which include the set of concepts needed to describe a domain at a certain level of abstraction, together with the relationships existing between them. Model transformations [22], commonly described as meta-model mappings, enable the automatic transformation and evolution of models into (1) other models, defined at either the same (horizontal transformation) or different (vertical transformation) levels of abstraction; and (2) any given textual format (e.g. code). The key characteristic to MDSD success lies in the fact that models can work in the *problem space* rather than in the *solution space*, e.g. source code. Compared to traditional object oriented technologies, MDSD replaces objects by models, and model transformations appear as a powerful mechanism for incremental and automatic software development [12].

Our objective is to provide a flexible approach that reconciles component-based design and application execution with real-time analysis. For this purpose, we have carefully designed a software framework that provides a configurable run-time support for managing component distribution and concurrency. The chapter also describes the main characteristics of such framework, together with the design forces and design patterns that have been used. The development approach and tool-chain are illustrated by modeling a simplified version of the well-known "Cruise Controller Development" [19]. The presented case study comprises the architectural design, the deployment configuration, and the temporal analysis of the final application with Cheddar [28], which is a real-time scheduling tool, designed for checking task temporal constraints of a real-time system.

2 Overall Approach of the Proposed Development Process

The three abstraction levels that comprise the proposed development approach are supported, respectively, by a language for modeling component-based applications, a component framework implemented in C++, and the Cheddar analysis tool. All these tools are integrated and supported by a MDSD tool-chain that enables models to smoothly evolve from components to objects and analysis models. The development approach is based on the particular interpretation of the MDSD approach offered by the Model-driven architecture (MDA) [23] standard. In MDA, Platform-Independent Models are created at the level of abstraction provided by components, and Platform-Specific Models are supported by an object oriented framework, entitled *FraCC* (Framework for Concurrent Components) and implemented in C++, which provides platform-specific run-time support. The evolution of the application through the different abstraction levels is automatically performed by means of model transformations. Obviously, this approach can be followed using other framework, providing it fulfills the application requirements. In the field of RTS, there have been very promising results with the MDA approach. Significant examples include the Artist Design Network of Excellence in Embedded System Design [2] and the OpenEmbeDD project [25].

Model transformations enable the automatic evolution of models into other models or into executable code. But transformations are complex software artifacts, difficult to understand, develop and maintain. Moreover, model transformations have a non-modular structure that prevents them from being reused (totally or partially) in systems that may have similar requirements. The use of frameworks reduce the complexity of model transformations, since they have only to specialize their hot-spots, not to generate the whole application, and thus transformation maintenance and evolution is dramatically simplified. As a side effect, MDA can help simplifying the use of frameworks by hiding the complexity of their specialization mechanisms, as stated in [1]. In addition, the use of software frameworks for the PSM level offers additional advantages, namely: (1) they are normally designed for fulfilling the non-functional requirements of the application domain they target; and (2) they can facilitate final application reconfiguration, provided that they have tools for that purpose. On the other side, the framework implementation is a very time-consuming task, making its development only advisable when it can be reused in many similar applications.

In the proposed development approach, we distinguish three roles: that of framework developer, that of MDA supporter, and that of application developer. These roles can be played by the same or different persons or teams. This article focuses on the application developer role, describing the tools and artifacts he/she can use to develop component-based applications and analyze their temporal behavior. Starting from a set of requirements (functional and non-functional), the application developer (1) designs the specific application using an architectural component-oriented modeling language, (2) he/she then executes a model transformations in order to generate the application, and (3) he/she can use the configuration tools provided by *FraCC* to make further modifications to the generated application, thus configuring

its deployment. In addition, models enable early validation and verification of application properties, while other properties cannot be verified until the final implementation is obtained. Our purpose is twofold: to lessen development times and prototype testing, by using a MDA development environment, and, on the other hand, to analyze the application as soon as possible.

3 Related Work

In this section, we will briefly review some of the most relevant works related to the technologies used in development approach presented in this book chapter: component-based design and component models, analysis tools, frameworks and pattern languages.

Among general-purpose component models, we may cite Fractal [8], the CORBA component model [24], KobrA [3], SOFA 2.0 [9], SaveCCM [13], and ROBOCOP [20], among others. Fractal, SOFA, SaveCCM and ROBOCOP provide different kind of ADLs to describe the application architecture, and a code generation facility that generates the application skeleton in Java or C/C++, which must be later completed by the developer. The CORBA CCM was developed to build component-based applications over the CORBA communication middleware. It provides an IDL to generate the external structure of the components. KobrA is one of the most popular proposals, in which a set of principles is defined to describe and decompose a software system following a downstream approach based on architectural components. But in all cases the implementation of the code and structure of the component is still completely dependent on the developer. A complete and updated classification of component models can be found in [14], where the authors propose a classification framework for studying component models from three perspectives, namely *Lifecycle*, *Construction* and *Extra-Functional Properties*.

Regarding analysis tools, the Spin model checker [6] is a widely used software tool for specifying and verifying concurrent and distributed systems that uses linear temporal logic for correctness specifications. UPPAAL [5] is a toolbox for verifying RTS, which provides modeling and query languages. UPPAAL is designed to verify systems that can be modeled as networks of timed automata [7] extended with integer variables, structured data types, user defined functions, and channel synchronization. Cheddar [28] is a free real-time scheduling tool, designed for checking temporal constraints of an RTS. MAST [21] defines a model to describe the timing behavior of RTS designed to be analyzable via schedulability analysis techniques, and a set of tools to perform such analysis. It can also inform the user, via sensitivity analysis, how far or close is the system from meeting its timing requirements.

Frameworks are one of the most reused software artifacts, and their development has been widely studied [17]. New, more general and innovative proposals have recently appeared in the literature, focusing on the development and use of frameworks for software systems development in general. In [16], the authors propose a method for specializing object-oriented frameworks by using design patterns, which

provide a design fragment for the system as a whole. A design fragment, then, is a proven solution to the way the program should interact with the framework in order to perform a function. A conceptual and methodological framework for the definition and use of framework-specific modeling languages is described in [1]. These languages embed the specific features of a domain, as offered by the associated framework, and thus facilitate developers the use of such frameworks.

As Buschmann et al. [10] state, not all domains of software are yet addressed by patterns. However, the following domains are considered targets to be addressed following a pattern-language based development: service-oriented architectures, distributed real-time and embedded systems, Web 2.0 applications, software architecture and, mobile and pervasive systems. The research interest in the real-time system domain is incipient and the literature is still in the form of research articles. A taxonomy of distributed RT and embedded system design patterns is described in [15], allowing the reader to understand how patterns can fit together to form a complete application.

4 Component Model and Real-Time Execution Platform

This section describes the language we defined in order to model component-based applications with real-time requirements, together with the platform that provides the required run-time support for executing the applications. The modeling language has been developed and integrated in the free and open source Eclipse platform, while the execution platform is a component-based framework, programmed in C++. The main characteristics of the former are described in Sect. 4.1, the supporting tool-chain is shown in Sect. 5, and the chosen sample case study, the cruise control, is shown in Sect. 6. The most important and relevant development details of the implementation framework (the platform) are described in Sect. 4.2.

4.1 Modeling Primitives for Real-Time Component-Based Applications

Modeling component-based applications with real-time requirements require the definition of several modeling concepts, at different levels of abstractions, as shown in Fig. 1. Firstly, it is necessary to model the building blocks with which to make applications, the components, in such a way that real-time constraints can also be modeled. Secondly, new applications should be built by reusing and composing already defined components. And finally, it is necessary that the supporting run-time platform enables the user to select the number of processes and threads in which the components should be executed. All these concepts are present in the developed modeling language, organized in the following packages:

Component modeling: We adopt the classical definition, where components are units that encapsulate their state and behavior, and which communicate only through their ports. The messages components can exchange are defined and typed by interfaces, which define the services the component requires/provides from/to the rest of the application components. In our approach we model component behavior by means of timed automata. Timed automata, which can be thought of as being finite state—machines with timed events and timed conditions, is a very suited formalism for modeling reactive systems with timing constraints. Component activities, that is, the code executed by a component depending on its current active state, are defined in terms of the services contained in the interfaces. Interface definition, component modeling, timed automata modeling, and activity definition are all performed in isolated models in order to maximize model reuse. We define a final binding model, where activities are linked to timed automata's states, timed automata are linked to components, and then timed automata's events and activities are linked to required and provided interfaces, respectively.

Application modeling: An application is a set of components (instances) that are connected through their ports, based on the compatibility of the interfaces required/provided in each end.

Deployment modeling: Components can be executed by different processes, that can assigned to different computational nodes. It is also possible to define the number of threads of every process. This part of the language enables users to decide the number of threads that will execute the component code, as well as the allocation of the computational load to each thread. The deployment model provides great flexibility to the approach, since different concurrency models (number of processes and threads, as well as the allocated computational load) can be defined for the same application without needing to change its architecture. This model is also the input model for generating the analysis file for Cheddar, as explained in Sect. 5.

Timed automata are the key artifacts of the language and modeling approach described in this book chapter, since they decide which code the component executes, and whether the component react to messages sent to it or not. They link structure with code (represented by activities). Also, timed automata regions define the unit of computational load assigned to threads in the *Deployment modeling* package, since in a given region there is one and only one active state which code should be executed by the component. Finally, activities represent logic units of work, that must be performed periodically or sporadically, depending on the component state. Activities in *FraCC* are programmed in C++ and then linked to the state in which they are executed. Activities only depend on the interface definitions, and therefore can be reused in several timed automata.

The described modeling language is embedded in the Eclipse tool-chain, as described in Sect. 5, while some screenshots of its use are shown in Sect. 6, where a cruise control system is developed by using the language and its associated tools.

4.2 Design Drivers and Pattern Language for a Flexible and Analyzable Execution Platform

Considering the previously described modeling elements, it is necessary to provide an execution environment that is consistent with the behavior described in the models. We have designed a component-based framework for which the main architectural drivers are:

AD1 Control over *concurrency policy*: number of processes and threads, thread spawning (static vs. dynamic policies), scheduling policy (fixed priority schedulers vs. dynamic priority scheduler), etc. Unlike most frameworks, these concurrency issues are very important in order to be later able to perform real-time analysis, and thus the framework should allow users to define them.

AD2 Control over the *allocation of activities to threads*, that is, control over the computational load assigned to each thread, since we consider the activity associated to a state as the minimum computational unit. The framework allows allocating all the activities to a single thread, allocating every activity to its own thread, or any combination. In any case, the framework ensures that only the activities belonging to active states are executed.

AD3 To avoid "hidden" code, that is, code which execution is outside the developer's control. The code that manages the framework is treated as "normal" user code, and therefore he can assign it to any thread.

AD4 Control over the communication mechanisms between components (synchronous or asynchronous).

AD5 Control over component distribution in different nodes.

The design and documentation of the framework was carried out using design patterns, which is a common practice in Software Engineering [11]. In order to describe the framework we will use Figs. 2 and 3. Figure 2 shows the pattern sequence that has been followed in order to meet the architectural drivers mentioned above, while Fig. 3 show the classes that fulfill the roles defined by the selected patterns. At this point, it is worth highlighting that the same patterns applied in a different order would have led to a very different design.

Among the aforementioned drivers, the main one is the ability to define any number of threads and control their computational load (architectural drivers AD1 and AD2). This computational load is mainly determined by the activities associated to the states of the timed automata. In order to achieve this goal, the COMMAND PROCESSOR architectural pattern [10] and its highly coupled COMMAND pattern [18] have been selected, and they were the firsts to be applied in the framework design, as shown in Fig. 2. The COMMAND PROCESSOR pattern separates service requests from their execution by defining a thread (the command processor) where the requests are managed as independent objects (the commands). These patterns impose no constraints over command subscription to threads, number of commands, concurrency scheme, etc. The roles defined by these two patterns are realized by the classes `ActivityProcessor` and `RegionActivity`, respectively (see Fig. 3).

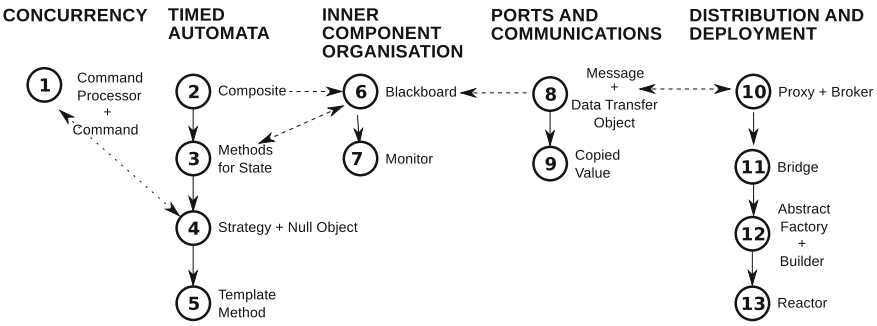


Fig. 2 Dependency relationships between the patterns considered in the framework development. Though the patterns are numbered, the design was iterative, and most of the patterns had to be revisited, leading to many design modifications

Another key aspect, related to AD3 and AD4, is to provide an object oriented implementation of timed automata compatible with the selected patterns for concurrency control, in order to integrate it in the scheme defined by the aforementioned COMMAND PROCESSOR pattern. It is also an aspect that has a great impact on the whole design, since timed automata model the behavior of the components. Timed automata are managed following the METHODS FOR STATES pattern [10], and the instances of the classes representing it are stored in a hash table. The class `Region` is an aggregate of `States`, and it is related to a subclass of `RegionActivity`, which defines how regions are managed. *FraCC* provides two concrete subclasses: `FsmManager` and `PortManager`. The former is in charge of (1) the local management of the region states (transition evaluation, state change, etc.), and (2) invoking the `StateActivity` of the region active state, while the latter is in charge of sending messages through output ports. The subclasses of `RegionActivity` constitute the link between concurrency control and timed automata implementation, since they are those that are allocated to command processors.

Conditions, transitions and events are modeled as separate classes, as shown in Fig. 3. `Condition` is an abstract class used to model transitions' conditions. It provides an abstract method to evaluate the condition. The only concrete subclass is `StateActiveCondition`, which tests whether a specific state is active. But the user can create his own subclasses to model other kind of conditions. The class `Transition` includes the source and target states, the event that triggers it, and a set of conditions vectors that must be evaluated to determine if the transition should be executed.

The next challenge is how to store and manage the component internal data, including all the states and activities mentioned above, the data received or that must be sent to other components, the transitions among states, event queues, etc. All these data is organized following the BLACKBOARD pattern. The idea behind the blackboard pattern is that a collection of different threads can work cooperatively on a common data structure. In this case, the threads are the command processors mentioned above.

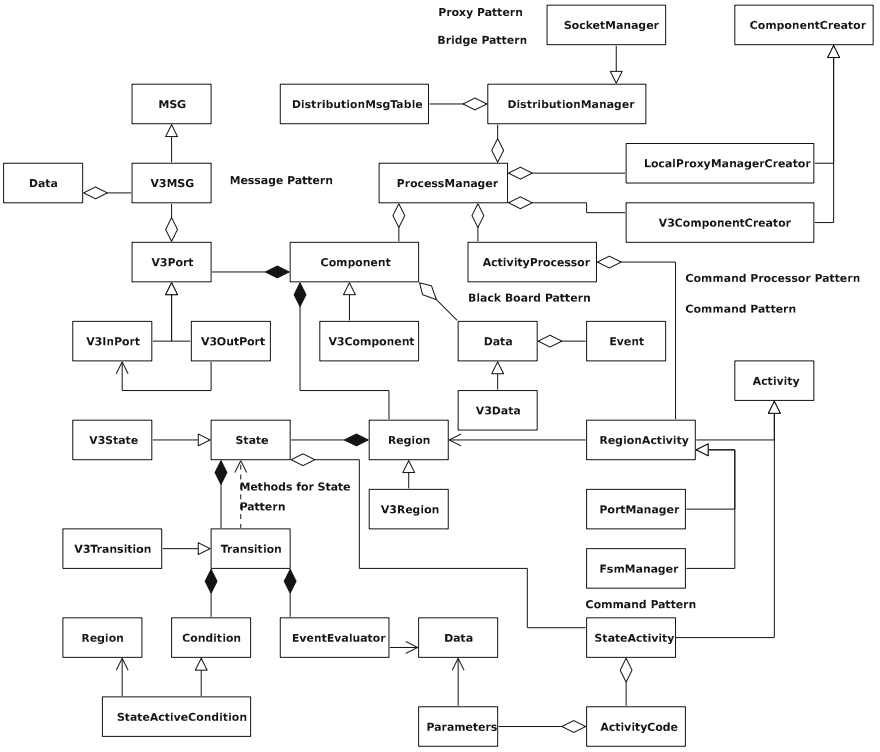


Fig. 3 Simplified class diagram of the developed framework showing some of the patterns involved in its design

The main liabilities of the BLACKBOARD pattern (i.e. difficulties for controlling and testing, as well as synchronization issues in concurrent threads) are mitigated by the fact that each component has its own blackboard, which maintains a relatively small amount of data. Besides, the data is organized in small hash tables. The roles defined by this pattern are realized by the classes `Data` and `V3Data`.

As shown in Fig. 2, the BLACKBOARD pattern serves as a joint point between timed automata and the input/output messages sent by components through their ports. Component ports and messages exchanged between them are modeled as separate classes. The classes representing these entities are the classes `V3Port` and `V3Msg`, shown in Fig. 3. The communication mechanism implemented by default in *FraCC* is the asynchronous without reply scheme, based on the exchange of messages following the MESSAGE pattern. In order to prevent the exchange of many small messages, we use the DATA TRANSFER OBJECT pattern to encapsulated in a single message all state information associated to a port interface, which is later serialized and sent through the port. Finally, since components encapsulate their inner state, we use the COPIED VALUE pattern to send copies of the relevant state information in each message. All these patterns are described in [10].

Component distribution is achieved by using Proxy components, which control the messages exchanged between components deployed into different nodes. These Proxy components are "regular" components, in the sense that they have ports with provided/required interfaces, just like the rest of the components. However, they are created at run-time, as stated in the deployment model, by the `LocalProxyManagerCreator` class. *FraCC* encapsulates the communication protocol by means of the BRIDGE pattern, which enables the user to change the used protocol. We currently support only the TCP protocol, embodied in the `SocketManager` class, but other implementations are also possible.

5 Description of the MDSD Tool-Chain: Modeling, Deployment and Analysis of Applications

A scheme of the different kind of software artifacts (models, meta-models, model transformations and tools) involved in the proposed development process is shown in Figs. 4, 5 and 6. A screenshot of some of the editors and models developed for the Eclipse-based tool-chain are shown in these figures, which are directly related to the different abstraction levels shown in Fig. 1, layer M2.

It is worth remembering that the first step of the proposed approach is to design the component-based application starting from its requirements. Applications in *FraCC* are built by connecting components imported from existing libraries. If the required components are not present in any of the available libraries, they should be firstly created and added to a library. Figure 4 summarizes the development process of this first step, as well as the modeling elements (as described in the previous section) and tools the developer uses. The Eclipse screenshots shown on the bottom of Fig. 4 correspond to a new component definition (on the left), and to the definition of a new application by connecting imported components (on the right).

Once the *FraCC* models have been created, the second step is to define how the application is going to be deployed. The models and tools involved in this second step are shown in Fig. 5. Starting from the *FraCC* models, an ATL transformation generates a default deployment model, which describes how the components are deployed into computational distributed nodes and concurrent processes. The developer can modify this model by using the deployment editor in order to fulfill application requirements, creating the final deployment model.

From the deployment model, the developer can select either to launch and execute the application by using the runtime support of *FraCC* or to generate analysis models (see Fig. 6). Particularly, we have developed a model transformation that generates a file for analyzing the schedulability of the *FraCC* application using Cheddar. Nevertheless, the process is flexible enough to allow the generation of analysis models for other tools. The separation between architecture and deployment enables the easy generation and analysis of different deployment strategies, without modifying the application architecture.

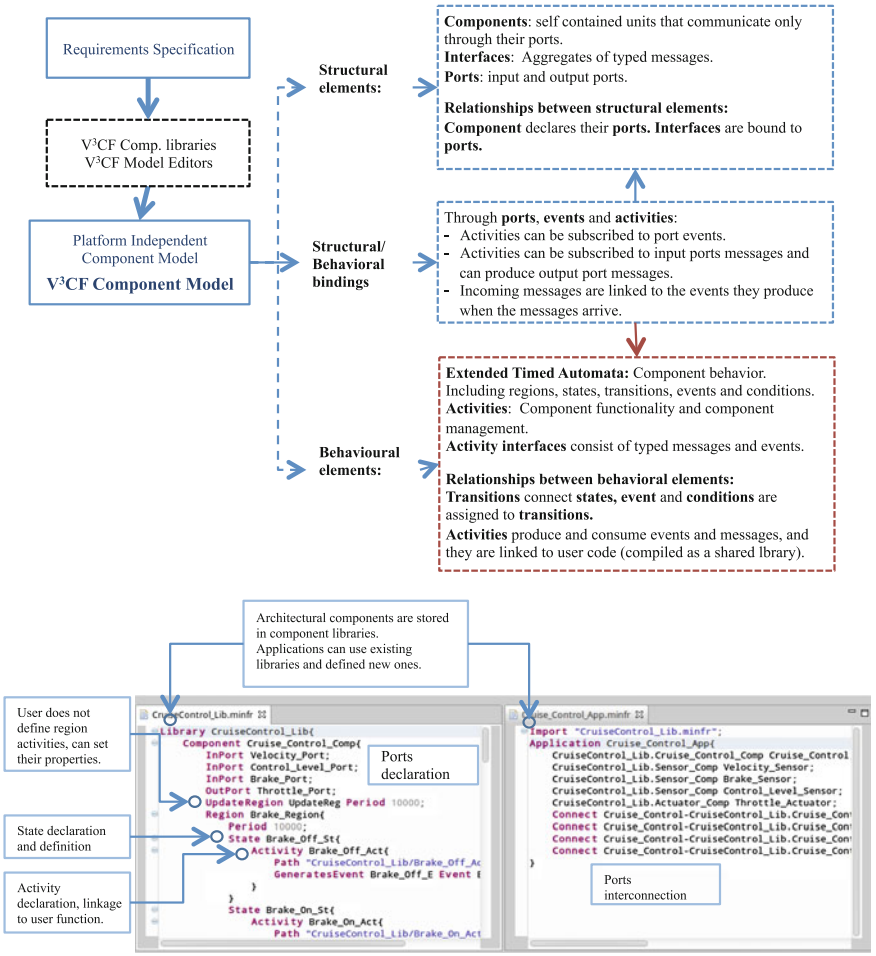


Fig. 4 Architectural software components for application design: artifacts, models, and eclipse tool-chain screenshots

It should be highlighted that *FraCC* does not give any guidance as to the number of threads that have to be created or how activities should be assigned to them, but it provides the necessary mechanisms to enable the user to choose the appropriate heuristic methods, for example the ones defined in [19]. Both the number of threads as well as the allocation of `RegionActivitys` to them can be done arbitrary, but the main objective should be "ensure application schedulability". For instance, a heuristic we normally use in our applications is to assign to the same thread `RegionActivitys` that have similar periods.

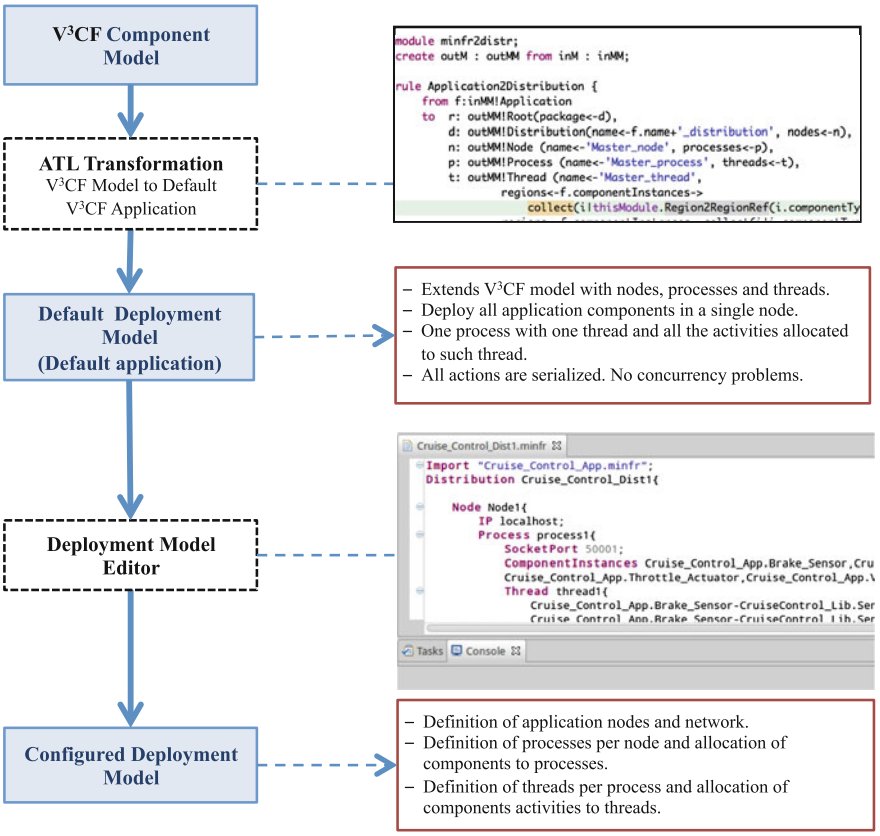


Fig. 5 Configuration of application deployment: artifacts, models, and eclipse tool-chain screenshots

6 Case Study: Development of a Cruise Controller

The case study that illustrates the use of *FraCC* and its associated tool-chain is a simplified version of the well-known "Cruise Controller Development" [19]. The original case study includes calibration and monitoring functions, which are not taken into account in the current example, since we decided to include only those functional requirements directly related to real-time system development. Besides, unlike the original solution, which is object-oriented, we develop a component-based one.

The cruise control system is in charge of automatically controlling the speed of a vehicle. The elements involved in the system are the brake and accelerator pedals, and a control level. This level has tree switch positions: ACCEL, RESUME, and OFF. The required cruise control functions are:

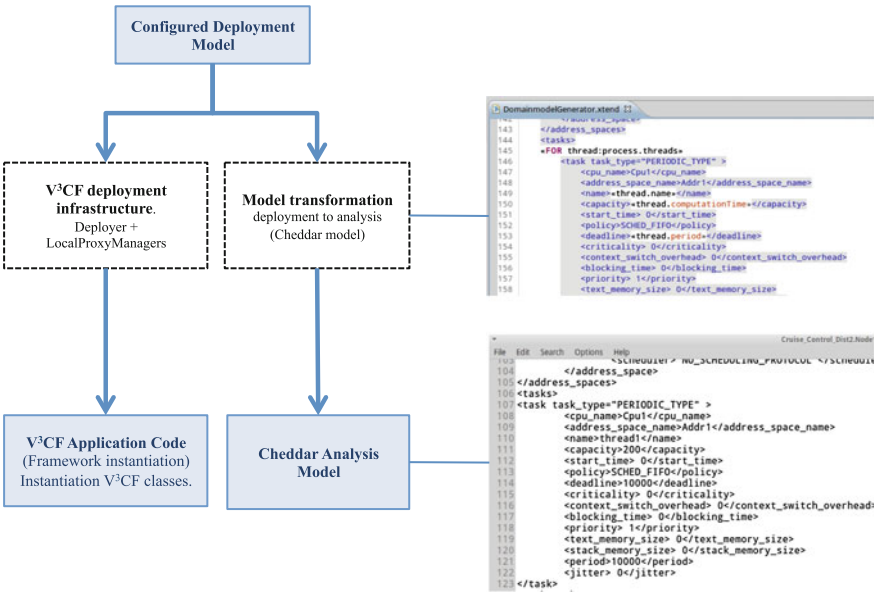


Fig. 6 Application and analysis models: artifacts, models, and eclipse tool-chain screenshots

ACCEL: with the cruise control level held in this position, the car accelerates without using the accelerator pedal. After releasing the level, the reached speed is maintained (referred to as the "cruising speed") and also "memorized".

OFF: by moving the control level to the OFF position, the cruise control is switched off, independently of the driving or operating condition. The cruise control is automatically switched off if the brake pedal is pressed.

RESUME: by switching the level to the RESUME position, the last "memorized" speed can be resumed. The "memorized" speed is canceled when the car engine is turned off.

6.1 Architecture of a Possible Solution

Due to their extension, it is not possible to show in a single figure all the components plus the timed automata that model their behavior. Therefore, we will first show the complete application architecture (see Fig. 7), while the rest of the timed automata will be progressively introduced.

As shown in Fig. 7, the cruise control is configured as a centralized application, comprising five components. Four of them encapsulate hardware access (*Brake_Sensor, Velocity_Sensor, Control_Level, Throttle_Actuator*), while the fifth

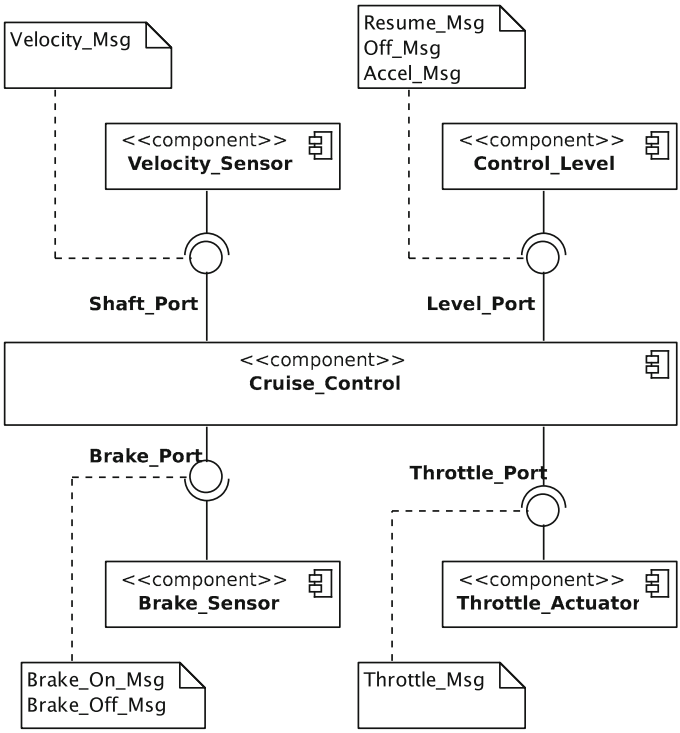


Fig. 7 Architecture of the cruise control application. Interface messages are shown as comments

one (*Cruise_Control*) models the whole control system and orchestrates the rest of the components.

The *Cruise_Control* component periodically receives messages from the sensor components, and, based on the data they provide and on the system state, calculates the action command and sends it to the *Throttle_Actuator* component. All the messages exchanged among components are always sent through the appropriate ports, as shown in Fig. 7. The *Cruise_Control* timed automata comprises three orthogonal regions: *Brake_Region, Control_Level_Region*, and *Cruise_Control_Region*, as shown in Figs. 8 and 9, respectively. This last region comprises the following states:

Initial state. When the driver turns the engine on, the region enters the initial state. The component remains in this state as long as no attempt is made to engage cruise control. In initial state, unlike Crusing_Off state, there is no previously stored cruising speed.

Crusing_Off state. When the driver either engages the level in the Off position (Off_E event) or presses the brake (Brake_On_E event), the cruise control is deactivated.

Accelerating state. When the driver engages the cruise control level in the ACCEL position (Accel_E event), the component enters into the Accelerating state and

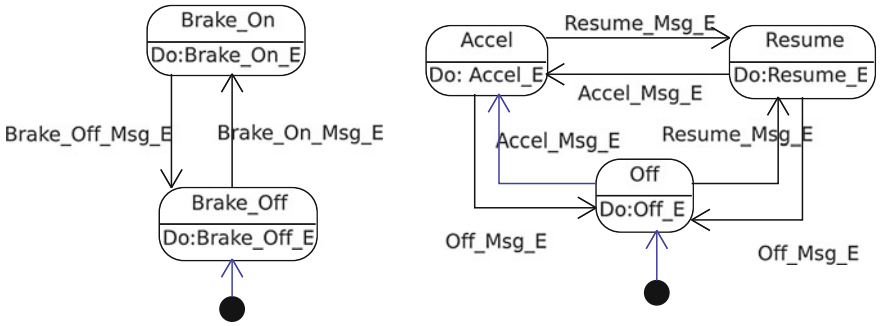


Fig. 8 Two of the regions that comprise the timed automata describing the behavior of the Cruise_Control component: *Brake_Region* on the *left* (stores the state of the car brake), and *Control_Level_Region* on the *right* (stores the state of the control level)

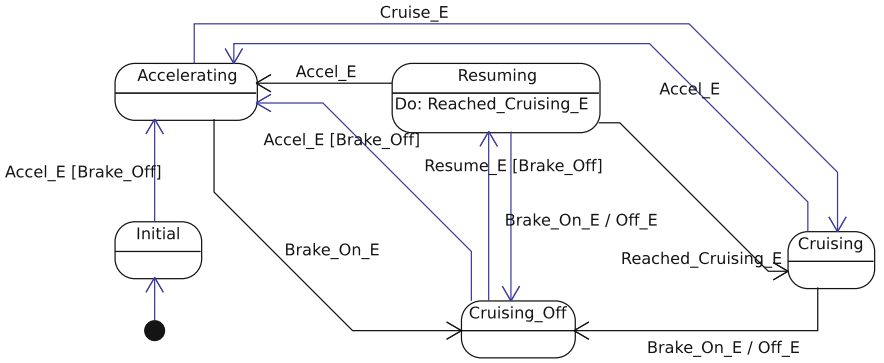


Fig. 9 The last region that comprise the timed automata describing the behavior of the Cruise_Control component

accelerates automatically, provided that the brake is not pressed (guard Brake_Off state).

Cruising state. When the driver releases the level (Cruise_E event), the current speed is saved as the cruising speed and the component enters the Cruising state, the car speed is automatically maintained at the cruising speed.

Resuming State. When the driver engages the level in the Resume position (Resume_E event), and providing the brake is not pressed, the car automatically accelerates or decelerates to the cruising speed. When the car reaches the desired speed, the region enters Cruising state (Reached_Crusing_E event).

Sensor components share the same timed automata, shown in Fig. 10 (left), though the activity that is periodically executed in each case is different. The same applies to the actuator component, shown in Fig. 10 (right). The activity associated to the state in each component is in charge of reading the sensor state, and then send messages to the *Cruise_Control* component.

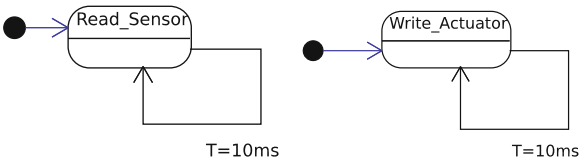


Fig. 10 Regions for controlling the sensors (*left*) and actuator (*right*) components

All the components describe above contain an additional region in their timed automata, not shown in the figures but present in the models, in charge of port management (as described in Sect. 4.2).

6.2 Architecture Deployment and Cheddar Analysis

Once the deployment model has been completed, a model-to-text XTend transformation (see Fig. 6) generates an analysis file for the Cheddar analysis tool. In order to perform the schedulability analysis, Cheddar requires the number of tasks, their temporal characteristics (WCET and period), and the number of shared resources of the application. The *Thread*s of the deployment model are directly transformed into Cheddar tasks, but shared resources must be derived from the deployment model, given *FraCC*'s memory structure and the assignment of `RegionActivity`s to threads made in the deployment model. Only buffers that are accessed by `RegionActivity`s assigned to different threads should be protected.

As mentioned in Sect. 5, one of the main distinguishing features of *FraCC* is the separation between architecture and deployment, which makes it possible to test different deployments (number of computational nodes, number of concurrent processes and threads, as well as the computational load assigned to every thread and their timing properties) easily without needing to modify the architecture. Figure 11 shows the schedulability analysis results, performed with Cheddar, of three different deployments of the cruise control application.

7 Conclusions and Future Work

The work described in this chapter is part of a more general approach for the development of component-based application supported by MDA technologies. The described MDA tool-chain hinders the complexity of the development process and automates the generation of both the final application and the analysis models. From our experience with the use of MDA technologies, model transformations are perhaps the most complex MDA artifacts, both in their design and maintenance. The higher the conceptual gap between the source and target abstraction levels, the higher the

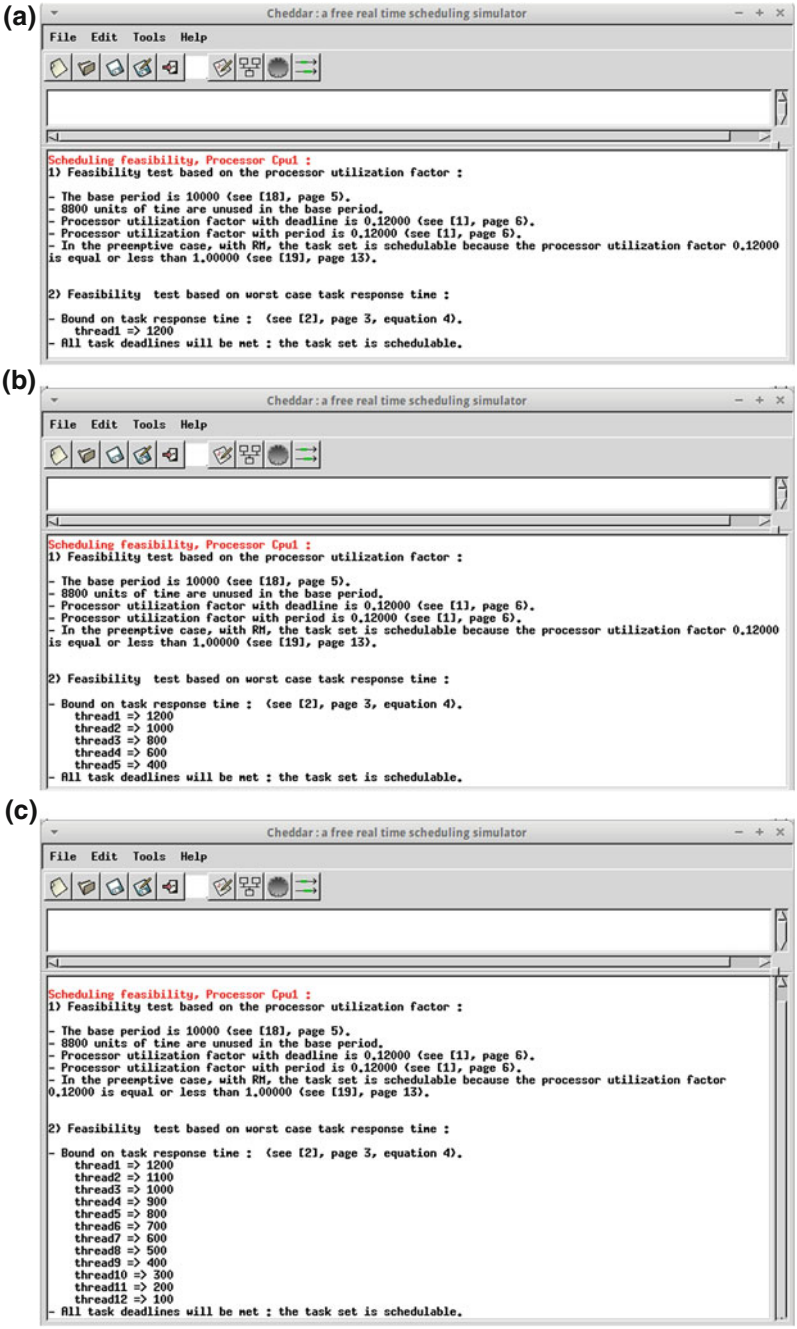


Fig. 11 Schedulability results of the Cheddar analysis tool for three different deployments of the cruise control application, as modeled in Fig. 7. **a** Analysis results of a deployment with one thread to which all `RegionActivitys` have been assigned to. **b** Analysis results of a deployment with five threads, to which `RegionActivitys` have been assigned to according to their periods. **c** Analysis results of a deployment with twelve threads, one for each `RegionActivity`

complexity of the transformation. Therefore, we decided to use a component framework as the target of the model transformations that generate the application code. This way, transformations have only to specialize its hot-spots, not to generate the whole code.

The development approach and tool-chain presented in this paper allow developers to use components as design units and threads an synchronization primitives as analysis units. *FraCC* is flexible enough to deal with changes in the application concurrency properties without changing the architectural design. By allowing developers to control the number of threads, their timing properties and computational load, he or she can analyze very different configurations before having to redesign the application. As a side effect, the separation between architecture and deployment enables the easy generation and analysis of different deployment strategies, without modifying the application architecture. It also facilitates component reuse, since the same functionality can be executed in different concurrency schemes.

It is also remarkable the way in which *FraCC* has been developed. The adoption of a pattern-driven approach has greatly facilitated the design of such framework. In addition, the selected patterns have been described like a pattern sequence. The design was iterative, and most of the patterns had to be revisited, leading to many design modifications.

Regarding future work, we are currently working on porting MinFr to non x86-based platforms, mainly 32-bits micro-controllers, and developing a reporting tool that presents the user with different deployment alternatives that optimize certain parameters, like number of threads, shared resources, communication bandwidth, etc. In the long term, we would like to integrate more complex analysis tools, like Uppal and Mast, as well as to use a third party component-based framework, being Robocop the most suitable alternative for our necessities. We are also working on generating input models for analysis tools compliant with the UML MARTE profile from instances of the framework.

Acknowledgments This work has been partially supported by the Spanish CICYT Project EXPLORE (ref. TIN2009-08572), the Séneca Project MISSION-SICUVA (ref. 15374/PI/10), and the Spanish MEC FPU Program (grant AP2009-5083).

References

1. K. Antkiewicz, M. Czarnecki, M. Stephan, Engineering of framework-specific modeling languages. IEEE Trans. Softw. Eng. **35**(6), 795–824 (2009)
2. Artist-ESD, *Artistdesign—European network of excellence on embedded systems design*, 2008–2011
3. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wst, J. Zettel, *Component-based Product Line Engineering with UML* (A-W Prof, Boston, 2001)
4. C. Baier, J. Katoen, *Principles of Model Checking* (The MIT Press, Cambridge, 2008)

5. G. Behrmann, K. Larsen, O. Moller, A. David, P. Pettersson, Y. Wang, Uppaal—present and future, in *Proceedings of the 40th IEEE Conference on Decision and Control*, vol. 3 (2001), pp. 2881–2886
6. M. Ben-Ari, *Principles of the Spin Model Checker* (Springer, Berlin, 2008)
7. J. Bengtsson, W. Yi, in *Timed Automata: Semantics, Algorithms and Tools*. Lectures on concurrency and Petri nets, vol. 3098 of LNCS (Springer, Berlin, 2004) pp. 87–124
8. G. Blair, T. Coupaye, J. Stefani, (eds.) Component-based architecture: the Fractal initiative. Ann. Telecommun. **64** (2009), Springer
9. T. Bures, P. Hnetynka, F. Plasil. Runtime concepts of hierarchical software components. Int. J. Comput. Inf. Sci. **8**, 454–463, (2007)
10. F. Buschmann, K. Henney, D.C. Schmidt, *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing* (Wiley, New York, 2007)
11. F. Buschmann, K. Henney, D. Schmidt, *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages* (Wiley, New York, 2007)
12. J. Bézivin, On the unification power of models. J. Syst. Softw. **4**(2), 171–188 (2005)
13. J. Carlson, P. Håkansson, J. Petterssonb, SaveCCM: an analysable component model for real-time systems. Electron. Notes Theoret. Comput. Sci. **160**, 127–140 (2006)
14. I. Crnkovic, S. Sentilles, A. Vulgarakis, M.R.V. Chaudron, A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2011)
15. L. Dipippo, C. Gill, *Design Patterns for Distributed Real-Time Embedded Systems. Real-Time* (Springer, Berlin, 2009)
16. G. Fairbanks, D. Garlan, W. Scherlis. Design fragments make using frameworks easier, in *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA 2006, ACM, (2006), pp. 75–88
17. M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design* (Wiley, New York, 1999)
18. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software* (A-W Prof, Boston, 1995)
19. H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML. Object Technology* (Addison-Wesley, Boston, 2000)
20. H. Maaskant, *Dynamic and Robust Streaming in and Between Connected Consumer-Electronic Devices*, volume 3 of Philips Research Book Series. Chapter A Robust Component Model for Consumer Electronic Products (Springer, Berlin, 2005), pp. 167–192
21. J. Medina, M. González-Harbour, J. Drake, Mast real-time view: A graphic uml tool for modeling object-oriented real-time systems, in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 245–256, Dec 2001
22. T. Mens, P. van Gorp, A taxonomy of model transformation. Electron. Notes Theoret. Comput. Sci. **152**, 125–142 (2006)
23. OMG, *Model driven architecture guide*, version v1.0.1, omg/2003-06-01, June 2003
24. OMG, *Corba component model*, v4.0, formal/2006-04-01, 2006
25. OpenEmbeDD, *Openembedd project, model driven engineering open-source platform for real-time & embedded systems*, 2008–2011
26. B. Selic, The pragmatics of model-driven development. IEEE Trans. Softw. Eng. **20**(5), 19–25 (2003)
27. M. Shaw, P. Clements, The golden age of software architecture. IEEE Softw. **23**(2), 31–39 (2006)
28. F. Singhoff, A. Plantec, P. Dissaux, J. Legrand, Investigating the usability of real-time scheduling theory with the cheddar project. J. Real Time Syst. **43**(3), 259–295 (2009)
29. C. Szyperski, *Component Software: Beyond Object-oriented Programming*, 2 edn. (A-W, Boston, 2002)

Automatic Development of Embedded Systems Using Model Driven Engineering and Compile-Time Virtualisation

Neil Audsley, Ian Gray, Dimitris Kolovos, Nikos Matragkas, Richard Paige and Leandro Soares Indrusiak

Abstract The architectures of modern embedded systems tend to be highly application-specific, containing features such as heterogeneous multicore processors, non-uniform memory architectures, custom function accelerators and on-chip networks. Furthermore, these systems are resource-constrained and are often deployed as part of safety-related systems. This necessitates the levels of certification and the use of designs that meet stringent non-functional requirements (such as timing or power). This chapter focusses upon new tools for the generation of software and hardware for modern embedded systems implemented using Java. The approach promotes rapid deployment and design space exploration, and is integrated into a fully model-driven toolflow that supports existing industrial practices. The presented approach allows the automatic deployment of architecture-neutral Java code over complex embedded architectures, with minimal overheads and a run-time support that is amenable to real-time analysis.

1 Introduction

Due to their application-specific nature, the architectures of modern embedded systems are commonly very different to that of more general-purpose platforms. Such systems contain non-standard features that are poorly supported by existing languages and development flows, which can make embedded design difficult, slow, and costly.

Good examples of this trend can be observed in recent smartphone devices. The Apple iPhone 3G, released in 2008, contained two main heterogenous processors (an application processor and a baseband processor), four different memory technologies of different speeds and sizes (DDR SDRAM, serial flash, NOR flash and SRAM),

N. Audsley (✉) · I. Gray · D. Kolovos · N. Matragkas · R. Paige · L. S. Indrusiak
Department of Computer Science, University of York, York, UK
e-mail: neil.audsley@york.ac.uk

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_2, © Springer-Verlag Berlin Heidelberg 2014

and a wide range of supplemental processing devices such as touchscreen controllers and power management controllers [8]. In later versions of the device the application processor itself became a heterogeneous, multicore, system-on-chip containing two ARM Cortex-A9 CPUs with a SIMD accelerator, dual core GPU, and dedicated image processing and audio processing cores. Developing software for such a system is extremely challenging and requires large amounts of low-level, hardware-specific software for each part of the system.

The difficulty of software development for complex architectures is compounded by the observation that many embedded systems are deployed in resource-constrained environments and so the efficiency of the final design is a top priority. Also, many embedded systems are real-time systems and so are required to be analysed and certified before deployment to ensure that they are fit for purpose.

This chapter discusses these problems in detail and considers existing solutions in Sect. 2. An approach is then presented that is part of the MADES project, an EU 7th Framework Project [40]. The MADES project uses model-driven techniques to seamlessly integrate model transformation (Sect. 3.2), software generation (Sect. 4) and hardware generation (Sect. 5) flows to promote rapid development, design space exploration, and increase the quality of the final systems. A case study is then presented in Sect. 6 to show how these tool flows work in practice. Finally, the chapter concludes in Sect. 7.

2 Background

This section will discuss the unique challenges of embedded development and some of the ways that they are currently addressed. Section 2.1 discusses the complex hardware architectures found in embedded systems, Sect. 2.2 discusses the problems faced by developers of safety-critical and high-integrity systems, and Sect. 2.3 describes industrial concerns.

2.1 Heterogenous Hardware Platforms

The hardware architectures of embedded systems are becoming increasingly non-standard and application specific. Large increases in on-chip transistor density coupled with relatively modest increases in maximum clock rates [20] have forced the exploration of multi-processor architectures with heterogenous processing components in order to meet increasing application performance requirements. Consequentially, many modern embedded systems target Multiprocessor Systems-on-Chip (MPSoCs)-based platforms. These architectures are a significant deviation from the homogeneous, uniprocessor platforms that have traditionally been the main component of embedded architectures.

Embedded architectures frequently contain multiple, *heterogeneous* processing elements [25], non-uniform memory structures [3], and non-standard communication facilities (e.g Network-on-Chip communications structures are used on the recent Tilera 64-core TILEPro64 processor [44] and the Intel 48-core Single-Chip Cloud Computer [26]). Embedded systems also make extensive use of application-specific hardware, such as DSP cores, function accelerators, or configurable processors [11]. For example, Texas Instruments' OMAP 5 range of devices [38] contain a dual-core ARM Cortex A15, two other smaller ARM cores, DSPs, and a GPU core.

The lack of a 'standard' architecture means that such systems are not well-supported by the standard toolchains and languages that have been previously developed. This is because the abstraction models of existing programming languages were not developed to cope such variety and variability of heterogeneous platforms. Early computer architectures were largely uniform and entirely static, consisting of a single processor with access to one contiguous block of memory. As a result, the abstraction layers of programming languages hid many architectural details to aid the programmer. This approach has been inherited by modern languages, which increasingly rely on the presence of middleware or a distributed operating system to allow the programmer access to hardware features and architectural mapping. Access to features such as complex memory or custom hardware can only be achieved though the use of abstraction-breaking techniques (link scripts, inline assembly, raw pointers etc.). These techniques are error-prone, difficult to port to new architectures, and hard to maintain. Also, on resource-limited embedded systems complex operating systems or middleware is infeasible.

2.2 Criticality

In addition to the problems described above, embedded systems are frequently deployed in safety-related (i.e. safety-critical) environments, thereby categorising them as hard real-time systems [6]. Such systems must be amenable to worst-case execution time analysis so that their worst-case timing behaviour can be identified and accounted for. This requires predictability at all stages of the design, from language choice (frequently a high-integrity subset such as Ravenscar Ada [5] or Java [24]) through a real-time OS (such as MARTE OS [34]) to real-time hardware features (such as the CAN bus, or SoCBUS [45]).

The heterogenous hardware of embedded systems can often make guaranteeing worst-case timing or resource use very difficult. Many hardware features have highly variable response times. For example, the response time for a cache is relatively low for a cache hit but very high for a cache miss. Characterising memory accesses as hits or misses at analysis time is an active area of timing analysis research [16, 33], made even harder when multi-level or shared caches are considered.

Once a suitable timing model of the hardware can be constructed that allows analysis, restrictions must be imposed onto the programming model that developers

can use in order to support timing analysis of the application software. The commonly used model [6] makes the following assumptions:

- The units of computation in the system are assigned a potentially dynamic priority level.
- At any given time the executing thread can be determined from the priorities in the system and the states of the threads. i.e. Earliest Deadline First scheduling states that the thread with the nearest deadline has the highest priority and should be executing, unless it is blocked.
- Priority inversion (deviations from the above point) in the final system can be prevented, or predicted and bounded.
- Threads contain code with bounded execution times. This implies bounds on loop iterations, predictable paths through functions, restrictions on expected input data, and limitations on exotic language features like code migration, dynamic dispatch, or reflection.
- Blocking throughout the system is bounded and deadlock free.

Finally, once predictable hardware and software are developed it is still necessary for the highest levels of certification (such as the avionics standard DO-178B) to demonstrate traceability from requirements to software elements. Currently this is not well supported by existing toolchains.

2.3 Industrial Applicability

Industry is generally reluctant to switch to new programming languages and toolchains as this imposes a drastically different development approach with implicit problems of risk, acceptance and difficulties with legacy systems. In general, existing industrial methodologies must be supported rather than supplanted. Model-driven engineering (MDE) is becoming more frequently used in industrial projects [29] and represents a common way of tackling the higher abstractions of modern embedded systems [18]. However, as with programming languages it is desirable to remain with existing modelling standards (such as SysML [43] or MARTE [30]) and tooling wherever possible. Another parallel with restricted programming languages is that UML and profiles like MARTE are very complex and there are many different ways to model the same concept, so restricted and more focussed subsets can help with productivity.

2.4 Summary

In summary, the following issues are observed:

- Embedded systems employ complex, heterogeneous, non-standard architectures.

- Such architectures are poorly supported by existing programming methodologies which tend to assume 'standard' hardware architectures.
- Embedded systems are frequently real-time or safety critical systems. This limits the programming model which can be used and the middleware or operating systems that can be deployed.
- Complex embedded architectures are frequently very difficult to analyse for worst-case timing behaviour.
- Industrial developers are reluctant to move to new tools or development methodologies due to concerns over use of legacy code, certification, trust in existing tools, and user familiarity.

3 Introduction to Model-Driven Engineering

The approaches introduced in this chapter will leverage Model-Driven Engineering (MDE) to attempt to mitigate some of the problems previously described. This section will introduce MDE, metamodels, and model transformations, and then describe the model transformation framework that is used throughout the work described by this chapter.

MDE is a software development paradigm, which aims to raise the level of abstraction in system specification and to increase the level of automation in system development. In MDE, models, which describe different aspects of the system at different levels of abstraction, are promoted to primary artifacts. As such, models "drive" the development process by being subjected to subsequent transformations until they reach a final state, where they are made executable, either by code generation or model interpretation.

MDE relies on two facts [21]. First, any kind of system can be represented by models and second, any model can be automatically processed by a set of operators. Since, models need to be understood and processed by machines, they need to conform to a metamodel. Metamodels are used as a typing system to provide precise semantics to the set of models they describe. Therefore, a metamodel is a model, which defines in a precise and unambiguous way a class of valid models. The metamodel describes the abstract syntax of a modelling language. The homogeneity of definition provided by metamodels enables engineers to apply operations on them such as transformations or comparisons in an automatic and generic way. Figure 1 illustrates the basic relations of conformance and representation between a system, a model and its corresponding metamodel.

3.1 Model Transformations

Model transformations play a key role in model-driven development. Czarnecki and Helsen [7] identify the following areas in which they are most applicable:

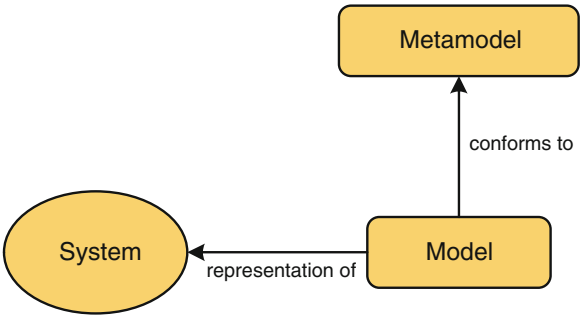


Fig. 1 Basic relations of representation and conformance in MDE (adapted from [21])

- Generating lower-level models and code from higher-level, more abstract models;
- Mapping between different models;
- Querying and extracting information from models;
- Refactoring models;
- Reverse engineering of abstract models from concrete ones.

Model transformations are computer programs, which define how one or more input models can be transformed into one or more output models. A model transformation is usually specified as a set of relations that must hold for a transformation to be successful. The input and output models of the transformation have to conform to a metamodel.

A model transformation is specified at the metamodel level and establishes a mapping between all the models, which conform to the input and output metamodels. Model transformations in MDE follow the model transformation pattern illustrated in Fig. 2. The execution of the rules of a transformation program results in the automatic creation of the target model from the source model. The transformation rules, as well as the source and target models conform to their corresponding metamodels. The transformation rules conform to the metamodel of the transformation language (i.e. its abstract syntax), the source model conforms to the source metamodel and the target model conforms to the target metamodel. At the top level of this layered architecture lies the meta-metamodel, to which all the other metamodels conform.

3.2 Epsilon Model Transformations

Model transformation languages are used to specify model transformations. In general, model transformations may be implemented in different ways, for example, by using a general purpose programming language or by using dedicated, domain specific model management languages.

In the context of MADES, the model transformation language used is the Epsilon Generation Language (EGL) [35], which is the model-to-text transformation

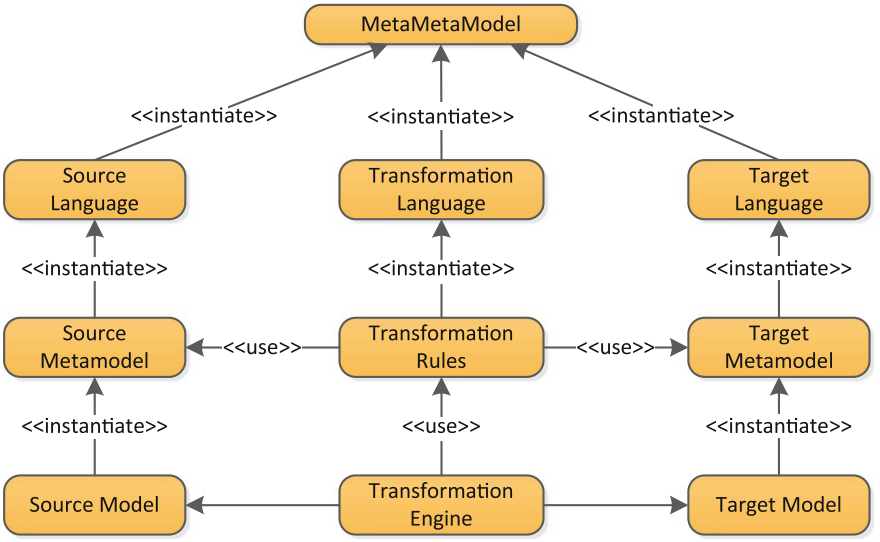


Fig. 2 Model transformation pattern [4]

language of the Epsilon framework [23]. Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is a family of consistent and interoperable, task-specific, programming languages which can be used to interact with models to perform common MDE tasks such as code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring.

Epsilon consolidates the common features of the various task-specific modelling languages in one base language and then develops the various model management languages atop it. The Epsilon Connectivity Layer (EMC) abstracts different modelling frameworks and enables the Epsilon task-specific languages to uniformly manage models of those frameworks. The architecture of the Epsilon framework is illustrated in Fig. 3.

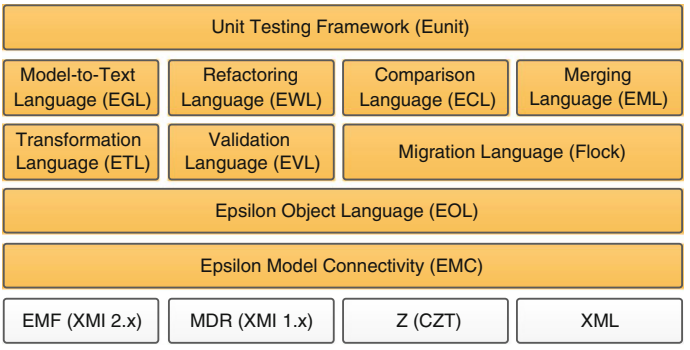


Fig. 3 Epsilon framework architecture

The approach proposed by this chapter is not dependent on the model management framework. However, Epsilon was preferred because of some of its unique features simplify the implementation activities. Such features include the support of Epsilon for interactive model transformations, the fine-grained traceability mechanism of EGL, as well as the framework's focus on reusability and modularity. Moreover, Epsilon is a mature model management framework with an active and large community.

4 Software Generation Using Compile-Time Virtualisation

Given the problems highlighted in Sect. 2, it can be seen that software development for many modern embedded systems is very challenging. Any solution to these problems must be industrially-acceptable so from the discussions in Sects. 2.3 and 2.2 the following requirements can be obtained:

- No new programming languages or tools because of certification requirements.
- No large runtime layers, or complex translated code.
- Integration with model-driven development to aid developers.

The MADES project therefore uses a model-driven approach which integrates a technique called Compile-Time Virtualisation (CTV) [14, 15]. Section 4.1 describes CTV and motivates its use while Sect. 4.2.3 describes how CTV is integrated into MADES.

4.1 Compile Time Virtualisation

Compile Time Virtualisation (CTV) is a source-to-source translation technique that aims to greatly simplify the development of software for embedded hardware architectures. It does this by integrating hardware virtualisation to hide the complexities of the underlying embedded architecture in a unique way that imposes minimal runtime overheads and is suitable for use in real-time environments. CTV allows the developer to write software for execution on a 'standard' desktop-style environment without having to consider the target platform. This architecturally-neutral input software is automatically translated to architecturally-specific output software that will execute correctly on the target hardware. The output software is supported by an automatically-generated, minimal-overhead, runtime that avoids the code size increase of standard middleware technologies (such as CORBA [32]) and run-time virtualisation-based systems (such as Java). CTV is a language-independent technique that can be applied to a range of source languages. It has currently been demonstrated in C [14] and Java [13]. The rest of this chapter will discuss CTV as it is applied to Java, but the approach is broadly the same in all languages.

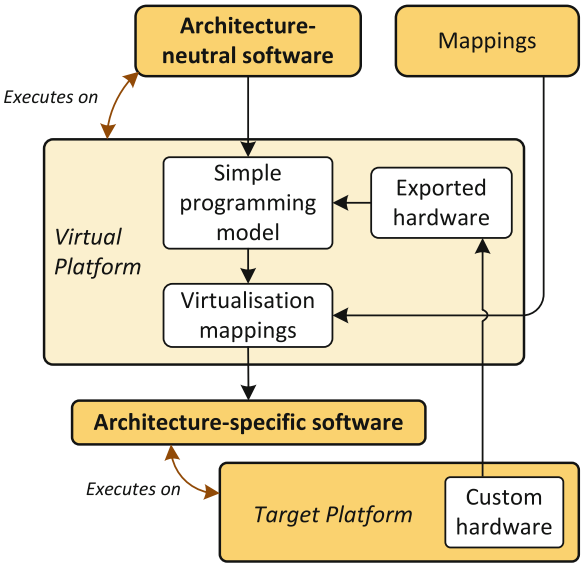


Fig. 4 Compile-time virtualisation introduces a virtual platform to make software development easier

CTV introduces a virtualisation layer over the target hardware, called the *Virtual Platform* (VP). This is shown in Fig. 4. The VP is a high-level view of the underlying hardware that presents the same programming model as the source language (in this case Java) to simplify development. For Java, it presents a homogeneous symmetric multiprocessing environment with a single monolithic shared memory, coherent caching, and a single uniform operating system. This is equivalent to a standard desktop computer running an operating system like Linux or Windows and is the environment in which Java's runtime expects to operate. *Therefore, the developer can write normal, architecture-independent Java code.*

As its name implies, the VP is a *compile-time only* construct, it does not exist at run-time. This is because the VP's virtualisation is implemented by a source-to-source translation layer that is guided by the virtualisation mappings (that map threads to CPUs and data to memory spaces). This can be seen in Fig. 5. The job of the source-to-source translation is to translate the architecturally-independent input software into architecturally-specific output code that will operate correctly on the target hardware, according to the provided mappings.

Unlike a standard run-time virtual machine, the virtualisation mappings are exposed to the programmer. This allows the programmer to influence the implementation of the code and achieve a better mapping onto the architecture. For example, by placing communicating threads on CPUs that are physically close to each other, or locating global data in appropriate memory spaces to minimise copying. Such design space exploration can be performed very rapidly because software can be moved throughout the target system without recoding.

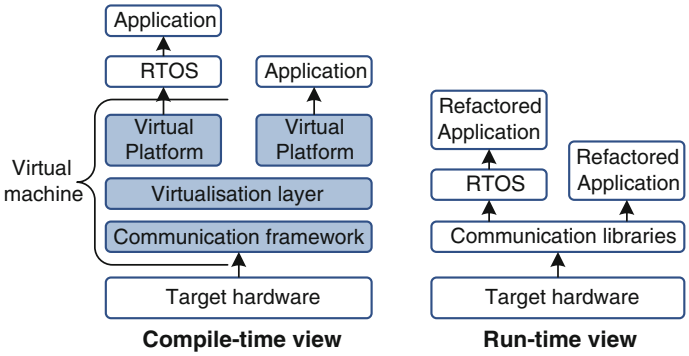


Fig. 5 Compile-time virtualisation hides complex hardware, but only at compile-time

Also in contrast to run-time virtual machines, custom hardware can be exported up to the programmer through the VP at design-time and presented in a form that is consistent with the source language's programming model, thereby allowing it to be effectively exploited by the programmer.

Moving the virtualisation to compile-time rather than run-time helps to reduce run-time overheads. Such overheads in a CTV system are small because all the work is done by the refactoring engine at compile-time. However, a consequence of applying the refactoring at compile-time is that all necessary analysis must be able to be performed offline. This means that certain aspects of the input language are restricted. However, as discussed in Sect. 2.2, in a real-time system such restrictions are already imposed (e.g. in the Ravenscar [5, 24] and SPARK [17] real-time language subsets). For more detail on this, see Sect. 4.2.2. In general, the principle is that:

> A system which is implemented using Compile-Time Virtualisation trades runtime flexibility for predictability and vastly reduced overheads.

For examples of how this trade off can reduce overheads, see Sect. 4.2.4.

Some additional benefits of the VP is that its use abstracts hardware changes from the software developer. The developer only has to target the VP rather than the actual hardware and if the hardware is changed at a later date, the same software can be retargeted without any recoding or porting effort. Similarly, because the VP is implemented to support development in existing languages, developers do not have to be trained to use a new language and existing legacy code can be more easily reused. Also, because the architecture-specific output code is still valid Java, no new compilers or tool need to be written. This is of vital importance to high-integrity systems that require the use of trusted compilers, linkers, and other tools.

The CTV approach is different to techniques such as Ptolemy II [9] which aim to provide new higher-level and more appropriate abstractions for programming complex systems. CTV is instead designed to allow existing languages and legacy code to be used to effectively target such systems through the use of very low-overhead virtualisation. The two different approaches can actually be complementary

and used together, with CTV used as a low-overhead intermediary to bring legacy code or legacy programming languages into an otherwise Ptolemy-defined system.

CTV is the name for the general technique. Section 4.2 will now discuss AnvilJ, the specific implementation of CTV that is implemented in the MADES project.

4.2 AnvilJ

Section 4.1 gave a broad overview of CTV. However, CTV is a language-independent technique that can be implemented to work with a range of input languages. In the MADES project the chosen source development language is Java (and its real-time variants [12, 24]), therefore MADES uses *AnvilJ*, a Java-based implementation of CTV that is described in the rest of this section. The AnvilJ system model is described in Sect. 4.2.1. Whilst AnvilJ can accept the majority of standard Java, a few restrictions must be imposed and these are enumerated in Sect. 4.2.2. The way that MADES integrates AnvilJ into its model-based design flow is discussed in Sects. 4.2.3 and 4.2.4 concludes with a discussion of how AnvilJ results in a system which displays minimal runtime overheads.

4.2.1 AnvilJ System Model

AnvilJ is an implementation of CTV for the Java programming language and its related subsets aimed at ensuring system predictability, such as the RTSJ. The AnvilJ system model is shown in Fig. 6. Its input is a single Java application modelled as containing two sets:

- **AnvilJ Threads**: A set of `static final` instances or descendants of `java.lang.Thread`.
- **AnvilJ Shared Instances**: A set of `static final` instances of any other class.

Collectively, AnvilJ Threads and Shared Instances are described using the umbrella term *AnvilJ Instances*. AnvilJ Instances are static throughout the lifetime of the system; they are created when the system starts and last until system shutdown.

An AnvilJ Instance may communicate with any other AnvilJ Instance, however the elements it has created may not communicate with the created elements of other AnvilJ Instances. This restriction allows the communication topology of the system to be determined at compile-time and the required runtime support to be reduced, as discussed later. This approach is particularly suited to embedded development because it mirrors many of the restrictions enforced by high-integrity and certification-focussed language subsets (such as the Ravenscar subsets of Ada [5] and Java [24] or the MISRA-C coding guidelines [41]).

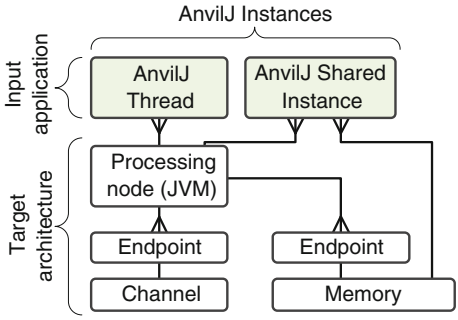


Fig. 6 The AnvilJ system model

In AnvilJ, the main unit of computation in the target hardware is the **processing node**. A processing node models a Real-Time Java Virtual Machine (JVM) [31] in the final system (or a standard JVM with accordingly reduced predictability). The Java specification does not define whether a multicore system should contain a single JVM for the entire system [1, 19] or one per core. Therefore AnvilJ models the JVMs, rather than the processors. The JVMs need not have similar performance characteristics or features. As with all CTV implementations, every AnvilJ Instance is mapped to exactly one node. AnvilJ Instances cannot migrate between processing nodes, but (if supported by the Java implementation) other instances can.

Nodes communicate using **channels**, which are the communication primitives of the target architecture. AnvilJ statically routes messages across the nodes of the system to present the totally-connected communications assumed by Java. The designer provides drivers for the channels of the system. **Memories** represent a contiguous logical address space and **endpoints** connect processing nodes to other hardware elements. Every AnvilJ Shared Instance must be mapped to either exactly one node (on the heap of the JVM), or exactly one memory where it will be available to all nodes connected to that memory.

This model is compile-time static—the number of AnvilJ Instances does not change at runtime. This is consistent with the standard restrictions that are imposed by most real-time programming models (as discussed in Sect. 2.2. For example, Ravenscar Ada [5] forbids all dynamic task allocation, whereas AnvilJ only forbids dynamic AnvilJ Instances. This is in contrast to systems like CORBA which adopt a "dynamic-default" approach in which runtime behaviour is limited only by the supported language features. Such systems support a rich runtime model but the resulting system can be heavyweight as they are forced to support features such as system-wide cache coherency, thread creation and migration or dynamic message routing, even if not required by the actual application. The approach of CTV is "static-default" in which the part of the application modelled is static. The restricted programming model promises less, but the amount of statically-available mapping information allows the required runtime support to be significantly reduced.

Not all instances of `java.lang.Thread` need to be modelled as an AnvilJ Instance. Equally, not all shared object instances need to be modelled at all. Enough should be modelled to fulfill the constraint that program instances created by an AnvilJ Thread t only communicate with other instances created by t, or another AnvilJ Instance.

4.2.2 Restrictions on Input Code

In order to be correctly refactored, AnvilJ input programs must be written to conform to a small set of restrictions which are detailed in this section. These restrictions are consistent with those required by existing real-time development processes (i.e. SPARK [17] or MISRA-C [41], see Sect. 2.2) and in most cases are less restrictive. They allow the system to operate with hugely reduced runtime overheads (see Sect. 4.2.4).

- AnvilJ threads and shared objects must be declared as `static final` fields. This means that the refactoring engine can determine at compile-time their location and number, which is not in general possible otherwise.
- All accesses to an AnvilJ object must directly refer to the field (using dot notation if the reference is in another class). It is forbidden to 'leak' a reference to an AnvilJ object, for example by returning it from a method, passing it to a method, or assigning it to a local variable of another class. Any of these actions will be checked by the refactoring engine and prevented.
- The arguments and return values of shared methods that are exported by an AnvilJ thread or shared object must implement `java.io.Serializable` interface.
- Threads on different nodes must only use other AnvilJ objects to communicate. Threads may perform any action that only affects the local JVM. However, if it calls methods or accesses fields with an instance on a different node that instance must be tagged as an AnvilJ Instance.

4.2.3 Integration with Model-Driven Engineering

To aid the use of AnvilJ, MADES integrates it directly into the model-driven engineering (MDE) flow of the project. This is not mandatory for AnvilJ, which can be used independently. In order to integrate AnvilJ it is necessary to provide the designer with a way of expressing a high-level view of the target hardware (in terms of the AnvilJ system model) and a high-level view of relevant parts of the input software. Not all the input software needs to be modelled, only the parts that are to be marked as AnvilJ Instances (Sect. 4.2.1). Also, the allocation of AnvilJ instances from the software model to the processing nodes of the hardware model must be provided.

This information is then translated from the designer's model into the form which is required by the AnvilJ tool. The translation is implemented using the Epsilon model transformation language, which is described in detail in Sect. 3.2. In the MADES framework, this information is provided by the designer through the use of 13 stereotypes which are applied to classes in the system model. These MADES stereotypes

Table 1 Brief description of the MADES stereotypes

Stereotype	Description
«mades_hardwareobject»	Superstereotype for all hardware stereotypes
«mades_clock»	Connected to «mades_processingnode» instances and «mades_channel» instances to denote a logical clock domain
«mades_channel»	A communication resource i.e. bus
«mades_ipcore»	Additional hardware i.e function accelerator
«mades_memory»	A single logical memory device
«mades_processingnode»	A computation element of the hardware platform. Commonly this is a single processor, but as described in Sect. 4.2.1, this corresponds to a JVM in the final system
«mades_endpoint»	Superstereotype of all endpoint stereotypes. Endpoints connect processing nodes to other hardware and provide more information about the connection. i.e. an ethernet endpoint may provide a MAC address
«mades_memorymedia»	Connects a «mades_processingnode» instance to a «mades_memory» instance
«mades_devicemedia»	Connects a «mades_processingnode» instance to a «mades_ipcore» instance
«mades_channelmedia»	Connects a «mades_processingnode» instance to a «mades_channel» instance
«mades_softwareobject»	Superstereotype for all software stereotypes
«mades_thread»	Represents an AnvilJ Thread
«mades_sharedobject»	Represents an AnvilJ Shared Instance

are described in Table 1. The modelling tool used in the MADES flow (Modelio [28]) supports two additional diagram types that use the MADES stereotypes; the detailed hardware specification and the detailed software specification. Allocations are performed with a standard allocation diagram. Working with these additional diagrams aids the designer because the MADES stereotypes can be automatically applied.

For a more detailed look at how the modelling is performed to integrate AnvilJ, Sect. 6 presents a case study that shows the development of a subcomponent of an automotive safety system.

4.2.4 Overheads

AnvilJ's static system model allows most of the required support to be implemented at compile-time, resulting in a small runtime support system, especially when compared with much larger (although more powerful) general-purpose frameworks. As will be shown in this section, the main overhead in an AnvilJ system is that of the Object Manager (OM). The OM is a microkernel which exists on every processing

Table 2 Class file sizes for OM features

Feature set	Approx. size (kB)
Thread creation and joining	5.7
Remote object locks	4.5
Shared methods	8.4
Sockets-based IComms (debug)	4.29
Full OM	34

node of the system and implements the AnvilJ system model. The OMs use a message-passing communications model to implement shared memory, locks, remote method calls etc.

The full version of the OM compiles to approximately 34 kB of class files including debugging and error information. However it is also possible to create smaller OMs which only support a subset of features for when the software mapped to a node does not require them. For example, if a node contains AnvilJ Shared Instances but no AnvilJ Threads then 5.7 kB of support for 'Thread creation and joining' can be removed. If none of the shared methods of a node are called then the shared methods subsection can be removed. The advantage of AnvilJ's offline analysis is that this can be done automatically each time, based on the exact input application and hardware mappings. Table 2 shows a breakdown of some of the feature sets of the OM and their respective code footprint.

Figure 7 compares this size to other similar systems. It should be noted that this comparison is provided purely to contextualise the size metric and demonstrate that AnvilJ's size is small, relative to related embedded frameworks. The other systems graphed, especially the CORBA ORBs, are built to support general-purpose, unseen software and consequentially are much more heavyweight.

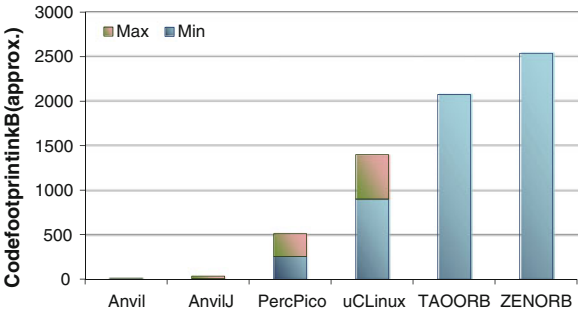


Fig. 7 The code footprint of the AnvilJ runtime compared to systems from a similar domains. Anvil is a C-based implementation of CTV, Perc Pico [2] implements safety-critical Java on systems without an OS, uClinux is a reduced Linux kernel for microprocessors with MMUs, and TAO [37] and ZEN [22] are Real-Time CORBA ORBs

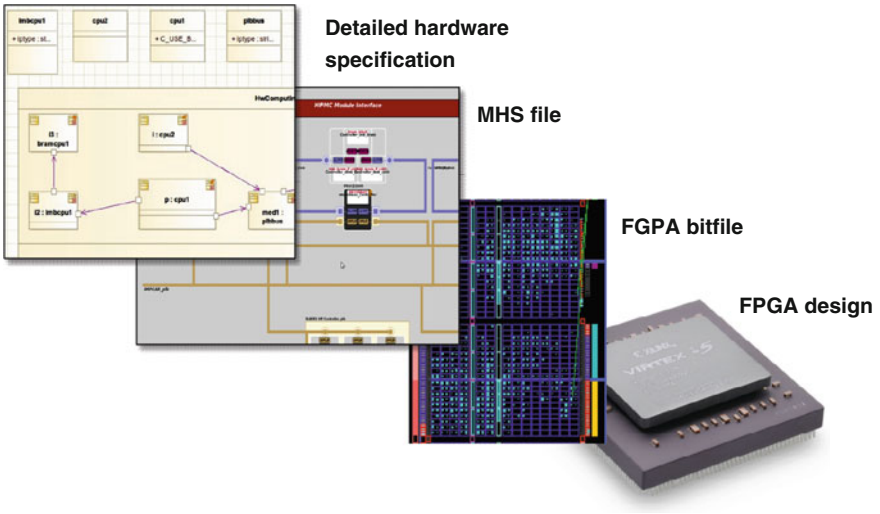


Fig. 8 The hardware generation flow

In addition to the small code size of the OM, its runtime memory footprint is also modest. The full OM in a desktop Linux-based system uses approximately 648 bytes of storage when idle, which increases as clients begin to use its features.

5 Hardware Generation Using Model Transformations

The MADES hardware generation flow transforms a detailed hardware specification diagram into an implementable hardware description. The generated hardware may be a complex, heterogeneous system with a non-uniform memory architecture but it is supported and programmed by the software generated by the code generation transformations described in Sect. 4.

In order to best demonstrate the flexibility of the hardware generation flow, the translations target Xilinx FPGAs. This is merely an implementation choice and does not reflect any part of the flow which inherently requires Xilinx devices and tools (or FPGAs in general). Other implementation structures can also be supported. The transformation outputs a Microprocessor Hardware Specification (MHS) file [46] which is passed to Xilinx Platgen, a tool that is part of Xilinx's Embedded Development Kit design tools [47]. Platgen is a tool which reads an MHS file and outputs VHDL [36] which can then implemented on the target FPGA. This flow is illustrated in Fig. 8.

The hardware generation flow is implemented using the Epsilon Generation Language (EGL) (see Sect. 3.2). There are three main benefits gained from generating hardware from the system model in this way:

- Very rapid prototyping and design space exploration can be achieved using this method due to the fact that hardware architectures can be constructed in the developer's modelling environment rather than vendor tools.
- MDE allows a vendor-neutral way of modelling and generating architectures. The same models could be used to target a wide range of FPGAs, ASICs, or even other hardware description languages like SystemC, however such an approach would not support the full flexibility of these systems.
- The same model is used as a source for both the software generation and hardware generation flows. These models share a consistent meta-model and so have related semantics. This gives confidence in the final design, because the software generation flow is refactoring code according to the same hardware model used by the hardware generation flow. In essence, the two flows 'meet in the middle' and support each other.

When creating the detailed hardware specification diagram, the hardware only needs to be modelled at a high level of abstraction. The platform is modelled as a class stereotyped with the stereotype «mades_architecture». Each detailed hardware specification contains exactly one such class. Properties in the «mades_architecture» stereotype are used to guide the software generation process by denoting the entry point class of the input application and allocating the initial `Main` thread to a processing node.

The details of the architecture are modelled with the MADES hardware stereotypes. Processing nodes («mades_processingnode») are the elements of computation in the platform. Each node supports a logical JVM. They communicate with other nodes through the use of channels. Nodes connect to channels using the «mades_channelmedia» endpoint stereotype. Memories («mades_memory») are data-storage elements and are connected to channels using «mades_memorymedia». Other hardware elements («mades_ipcore») are connected to channels through the use of the «mades_devicemedia» endpoint stereotype.

The top-level hardware stereotype «mades_hardwareobject» defines a property called `iptype`. This is passed to the hardware generation transformation to specify the type of hardware which should be instantiated. Further properties can also be passed depending on the value of `iptype`. For an example of this see the case study in Sect. 6.5.

Clock domains are modelled by classes stereotyped with the «mades_clock» stereotype. Clock synthesis is restricted by the capabilities of the implementation target and the IP cores used. A set of design rules are first checked using model verification to ensure that the design can be realised. These are:

- The total number of clock domains is not higher than the limit for the target FPGA.
- All communications across clock boundaries use an IP core that is capable of asynchronous signalling (such as a mailbox).
- All IP cores that require a clock are assigned one.

Each clock has a target frequency in the model and is implemented using the clock manager cores of the target FPGA. As with all FPGA design, the described

constraints are necessary but not sufficient conditions. During synthesis the design may use more clock routing resources than are available on the device, in which case the designer will have to use a more powerful FPGA or reduce the clock complexity of the design.

Currently, interfaces (IO with the outside world) have to be taken from the IP library or manually defined in VHDL or Verilog. It is not the aim of this approach to provide high-level synthesis of hardware description languages such as in Catapult-C [27] or Spec-C [10], although such approaches can be integrated by wrapping the generated core as an IP core for the Xilinx tools.

6 Case Study: Image Processing Subsystem

This section will present a case study to illustrate the benefits of the MADES Code Generation approach and show how CTV/AnvilJ is integrated into the design flow. This case study will detail the development of a subsection of an automotive safety system called the Car Collision Avoidance System (CCAS). The CCAS detects obstacles in front of the vehicle to which it is mounted and, if an imminent collision is detected, applies the brakes to slow the vehicle. In this case study we focus on a small part of the detection subsystem and show how the MADES code generation allows architecture-independent software to be generated to process the radar images without concern for the target platform. Multiple hardware architectures can be modelled and the software automatically deployed over auto-generated hardware.

Section 6.1 gives a block-level overview of the developed component and Sect. 6.2 discusses how the initial software is developed. The generation of the software and hardware models is covered in Sects. 6.3 and 6.4. The generation of the target hardware is detailed in Sect. 6.5 and finally Sect. 6.6 discusses deploying the software to the generated hardware.

6.1 Subsystem Overview

The developed subsystem takes images from the radar (or camera) and applies JPEG-style compression to reduce the size of the image and therefore reduce the demand on on-chip communications. Once reduced in size, the images are passed on to other parts of the system for feature extraction and similar algorithms. The block diagram of the subsystem is given in Fig. 9. The main stages of the subsystem are as follows:

- **Read Image**: Periodically reads images from the input to the system from a radar or camera.
- **DCT**: A Discrete Cosine Transformation moves the representation of the image from the spatial domain into the frequency domain.

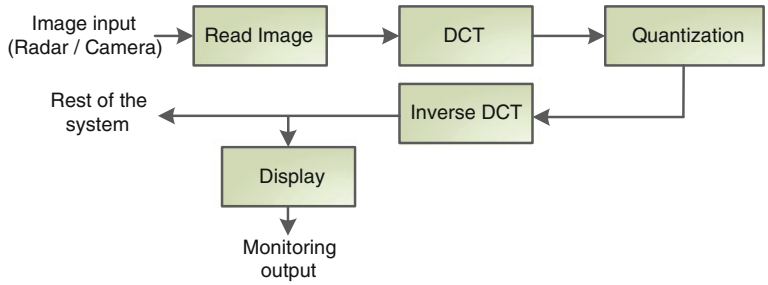


Fig. 9 Block diagram of the implemented subsystem. A monitoring output stage is included to allow verification of the subsystem during system development

- **Quantization**: Data in the frequency domain is selectively discarded to compress the image.
- **Inverse DCT**: Moves the image back into the spatial domain. The result is a (compressed) image that can be passed to the rest of the system, or optionally fed to a monitoring output stage.
- **Display**: Used for monitoring and debugging, the output stage uses a graphical user interface to display the image to the user.

6.2 Software Development

Developing the software for this subsystem is very simple when using AnvilJ because the developer can develop as if the code will execute on a standard desktop Java environment. However, the developer must observe the restrictions detailed in Sect. 4.2.2. Also it is not possible to develop the low-level drivers for the radar/camera input through AnvilJ directly, so for the purpose of testing and initial development stub drivers should be used that operate on the development platform. Final hardware interfacing must be done once deployment is underway as is normal practice.

The main restriction imposed by AnvilJ is that AnvilJ Instances must be static and only communicate through other AnvilJ Instances. This forces the developer to consider the structure of their code carefully, as is the case with all embedded development. The refactoring engine of AnvilJ allows the entire operation of the subsystem to be detailed using a single Java program, even though the final hardware platform may involve multiple heterogeneous processing elements. The code was structured as follows:

- Each block of the subsystem (see Fig. 9) is implemented as a `static final` thread. The threads are declared and started by a `Main` class that is responsible for initialising the system.
- Each thread contains internal state that holds images passed into it from the previous stage, and methods that allow the previous stage to pass in images to process.

Fig. 10 Example of the architecturally-neutral software operating in the development environment on a test image. The right-hand image is after processing

The thread processes images in its work queue, and passes completed images to the next thread.
- Each thread is designated as an AnvilJ Thread. This ensures that all communications in the system go through AnvilJ Instances.
- The output stage is designated an AnvilJ Shared Instance.
- Standard implementations of the DCT and Quantize stages are used from open source, freely-available code. This is one of the great advantages of AnvilJ in that often legacy code can be integrated easily.

Having created the software, its functionality can be verified immediately simply by executing the code in the development environment. It is not necessary to use simulators, cross-compilation or similar. The result of the software operating on a test image is shown in Fig. 10 and a listing of the `Main` class can be found in Fig. 11. Note that the listing is standard Java code, no extra-linguistic features are required.

6.3 Software Modelling

In a model-driven development flow, the architecture-independent software will be developed based on a software model. This model must be extended with a MADES 'Detailed Software Specification' diagram (detailed in the previous chapter) to inform the AnvilJ tool of the AnvilJ Instances that are present in the input software. This diagram links elements of the software model with the input software, using the concept of 'bindings'.

Bindings are a way of uniquely identifying source code elements (classes, instances, fields, methods etc.) and are defined by the Eclipse JDT project [39]. The developer obtains the binding for an AnvilJ Instance from the AnvilJ GUI and adds it to the `binding` property of the «mades_softwareobject» stereotype in the

```
public class Main {
  private final static int QUALITY = 20;

  public static final ReadThread readThread =
    new ReadThread();
  public static final DCTThread dctThread =
    new DCTThread(QUALITY);
  public static final QuantizeThread quantizeThread =
    new QuantizeThread(QUALITY);
  public static final OutputStage outputStage =
    new OutputStage(QUALITY);

  public static void main(String[] args) {
    readThread.start();
    dctThread.start();
    quantizeThread.start();

    readThread.join();
    dctThread.join();
    quantizeThread.join();
  }
}
```

Fig. 11 Listing of the `Main` class that initialises the implemented subsystem

software model. This links the instance in the detailed software specification diagram to the source code.

Figure 12 shows the completed detailed software specification diagram. The diagram is very simple as its only purpose is to add AnvilJ Elements to the software model and link them to the source code with binding keys. Note that the use of the «mades_thread» and «mades_sharedobject» stereotypes.

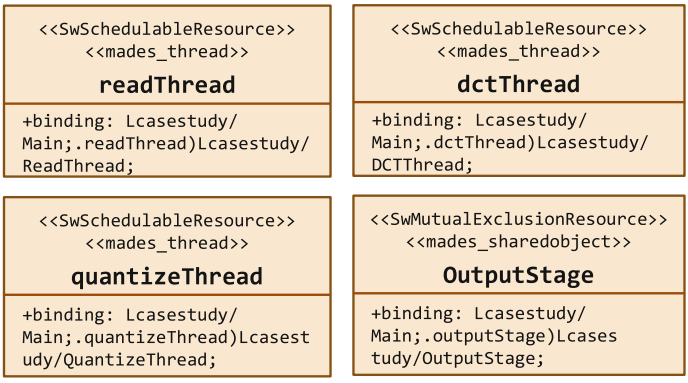


Fig. 12 The detailed software specification diagram for the case study subsystem

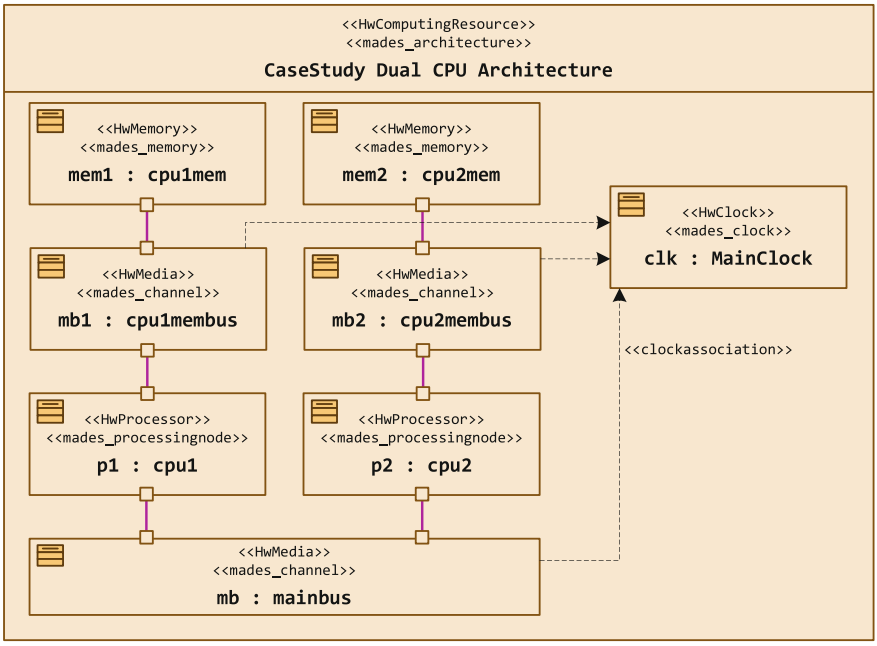


Fig. 13 The detailed hardware specification diagram for the case study target architecture. Not shown are properties in the classes that describe each hardware element in greater detail

6.4 Hardware Modelling

Having modelled the software, this section will now describe how the target hardware platform is modelled for AnvilJ integration. Recall that according to the AnvilJ system model from Sect. 4.2.1, it is only necessary for the hardware model to cover a high-level view of the capabilities of the target platform; in terms of processing nodes, memories, channels, and application-specific IP cores.

In this case study we will describe two target platforms and show how the same input software can be automatically deployed without recoding. The first presented architecture is a dual-processor system with a non-uniform memory architecture, shown in Fig. 13.

Once the detailed hardware model is complete, the hardware generation flow can be initiated.

6.5 Hardware Generation

The designer uses the MADES model transformations of Sect. 3.2 to transform the architecture modelled in Sect. 6.4 into an implementable hardware description.

```
PORT fpga_0_uart_RX_pin = fpga_0_uart_RX_pin, DIR = I
PORT fpga_0_uart_TX_pin = fpga_0_uart_TX_pin, DIR = O
PORT fpga_0_mainClk_pin = clock_mainClk, DIR = I,
  SIGIS = CLK
PORT fpga_0_sys_1_rst_pin = sys_rst_s, DIR = I,
  SIGIS = RST, RST_POLARITY = 1

BEGIN microblaze
 PARAMETER INSTANCE = cpu1
 PARAMETER C_USE_BARREL = 1
 PARAMETER HW_VER = 8.20.b
 PARAMETER C_DEBUG_ENABLED = 0
 BUS_INTERFACE DPLB = plbbus
 BUS_INTERFACE IPLB = plbbus
 PORT MB_RESET = mb_reset
 BUS_INTERFACE ILMB = cpu1_ilmb
 BUS_INTERFACE DLMB = cpu1_dlmb
END
```

Fig. 14 Fragment of the MHS generated by transforming the case study architecture of Fig. 13

As discussed previously in Sect. 5, the hardware generation flow targets Xilinx FPGAs and uses the Xilinx IP libraries from Xilinx Embedded Development Kit [47]. Accordingly, the hardware model must be augmented to include enough details to instantiate these IP cores. This is done by adding properties to the classes of the detailed hardware specification diagram. Full details of these properties are outside the scope of this chapter and are given in the MADES documentation [40].

Each of the MADES hardware stereotypes has a mandatory property called `iptype`. This is used by the Epsilon model transformation to inform it which Xilinx IP should be instantiated. Each supported IP has a set of *attributes* that may be also set from the model. For example, the `xps_uartlite` IP core is a serial transceiver and includes attributes such as `C_BAUDRATE` to set the expected baud rate and `C_USE_PARITY` to switch on or off the use of parity bits. The hardware generation flow checks for the presence of any mandatory attributes and warns the user if they are not present.

Once the model is completed with the required information, the user runs the hardware transformation to produce a Xilinx MHS file. A fragment of the MHS generated by transforming the case study architecture of Fig. 13 is shown in Fig. 14. This MHS file must be then converted into VHDL using the Xilinx design tools. For the purpose of this case study, the target will be a Xilinx Virtex 5 FPGA [48]. At the end of the implementation, an FPGA bitfile will be created which can then be programmed to the device for testing.

6.6 Code Deployment

After modelling the software and hardware, a deployment diagram can be created that maps instances from the detailed software specification to the detailed hardware specification. For this case study, the initial allocation will locate the image reading

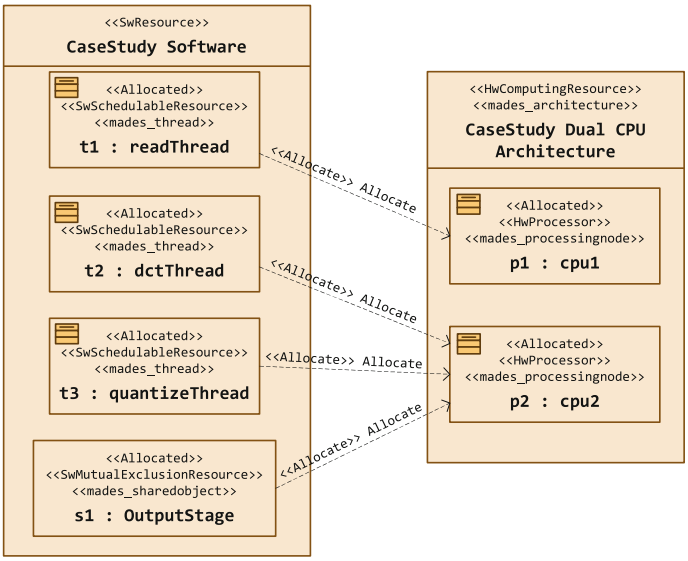


Fig. 15 An allocation diagram that deploys software from the detailed software specification diagram of Fig. 12 to the detailed hardware specification of Fig. 13

thread to CPU1 and all other threads to CPU2. The diagram that performs this allocation can be seen in Fig. 13 .

With the addition of the allocation diagram the is model is now complete, so it is exported in XMI format for use in the Eclipse IDE. Once imported to Eclipse, an Epsilon model transformation is used to create an AnvilJ architecture description. This file is created from the hardware, software, and allocation diagrams and is the input to the AnvilJ refactoring engine. It tells AnvilJ what the structure of the input software will be, which elements are AnvilJ Instances, the topology of the target platform, and how to place the AnvilJ Instances throughout the platform. Figure 16 shows the architecture description for the case study (Fig. 15).

Once an architecture description is created, the AnvilJ refactoring engine can be invoked at any time to refactor the architecturally-neutral Java application (an Eclipse project) into a set of architecturally-specific output programs, one for each processing node of the target platform as described in the hardware diagram. As the case study architecture has two processing nodes, two output projects will be created. AnvilJ is fully-integrated into the Eclipse Development Environment. After refactoring is complete, the output applications can be verified by executing both. AnvilJ's default implementation uses TCP sockets for inter-node communications, with the intent that developers replace this with the actual communications drivers of the target platform. However, this default allows immediate testing on standard networks. In this case, the two output projects coordinate as expected. The node with `ReadThread` reads example radar images and passes them to the other node *now running in a separate JVM* on which `quantizeThread` and `dctThread` process

```
<architecture name="CaseStudy Dual CPU Architecture"
    mainclass="casestudy.Main" maincpuid="0">
  <cpu name="cpu1" id="0">
    <thread binding="Lcasestudy/Main;.readThread)
      Lcasestudy/ReadThread;"/>
  </cpu>
  <cpu name="cpu2" id="1">
    <thread binding="Lcasestudy/Main;.quantizeThread)
      Lcasestudy/QuantizeThread;"/>
    <sharedobject binding="Lcasestudy/Main;.outputStage)
      Lcasestudy/OutputStage;"/>
    <thread binding="Lcasestudy/Main;.dctThread)
      Lcasestudy/DCTThread;"/>
  </cpu>
  <channel name="plbbus">
    <endpoint cpu="cpu1"/>
    <endpoint cpu="cpu2"/>
  </channel>
</architecture>
```

Fig. 16 The AnvilJ architecture description for the case study. Note the binding keys correlate with those of the software diagram in Fig. 12

them. `outputStage` displays the processed images. The two output binaries can be placed on separate networked computers with the same functional behaviour. The single input program has been automatically converted into a networked program according to the allocation diagram in the system model.

6.7 Analysis of Deployed Code

During refactoring, AnvilJ constructs a minimal runtime to support each output project and refactors the code to use this runtime. The refactoring engine reports all changes it is making to the input code for each output project so that the generated code can be traced back to the input code. These changes are very small and only occur at well-defined points. For example, these lines appear at the start of the `run()` method of the `Main` class of the input software:

```
readThread.start();
dctThread.start();
quantizeThread.start();
```

After refactoring this becomes:

```
//Instantiate the Object Manager for node id 0
anvilj.refactored.Globals.om = new anvilj.ObjectManager(
      new anvilj.Settings(true, false, false), 0,
      new anvilj.socketcomms.SocketComms(0),
      new anvilj.refactored.Architecture(),
      new anvilj.refactored.ThreadCreator(),
      new anvilj.refactored.SharedMessages(),
      new anvilj.refactored.Routing());
anvilj.refactored.Globals.om.start();

readThread.start();
//Start thread id 1 on node id 1
anvilj.refactored.Globals.om.startThread(1, 1);
//Start thread id 2 on node id 1
anvilj.refactored.Globals.om.startThread(1, 2);
```

This code sets up and initialises the Object Manager (OM, AnvilJ's runtime support) for the current node. The implementation of the OM is automatically generated in the `anvilj.refactored` package and is unique to each processing node of the final system. For example, the AnvilJ `Routing` object contains routes to the other nodes of the system with which this OM will need to communicate. Nodes that it does not communicate with are not detailed. If the code is updated then more or fewer routes may be added, but this will always be a minimal size. Routes are planned offline according to the detailed hardware specification diagram.

Note that two of the calls to `Thread.start()` have been rewritten by the refactoring engine to calls into the OM. This is because the threads `dctThread` and `quantizeThread` are allocated to another processing node, so they are started by calling into the AnvilJ runtime. The runtime sends a 'start thread' message to the processing node that hosts the given thread. The call to start thread `readThread` has not been translated, however, because it is allocated to the current node. If the allocation diagram is altered and AnvilJ is rerun, the refactored calls will change.

6.8 Retargeting for New Platforms

Retargeting the case study for a new architecture is simply a case of preparing a new detailed hardware specification diagram and amending the allocation diagram. Figure 17 shows a revised target architecture. This is the same as the original case study architecture (shown in Fig. 13) however a third processor has been added. The revised allocation diagram allocates the threads more evenly and can be seen in Fig. 18.

Once the model has been updated, it is re-exported as XMI and AnvilJ re-run. As the hardware diagram now contains three processing nodes, this produces three output projects with the AnvilJ Instances distributed as described by the allocation diagram. Once again, initial functional verification can be performed by executing the three output projects and observing that the functional behaviour is again identical.

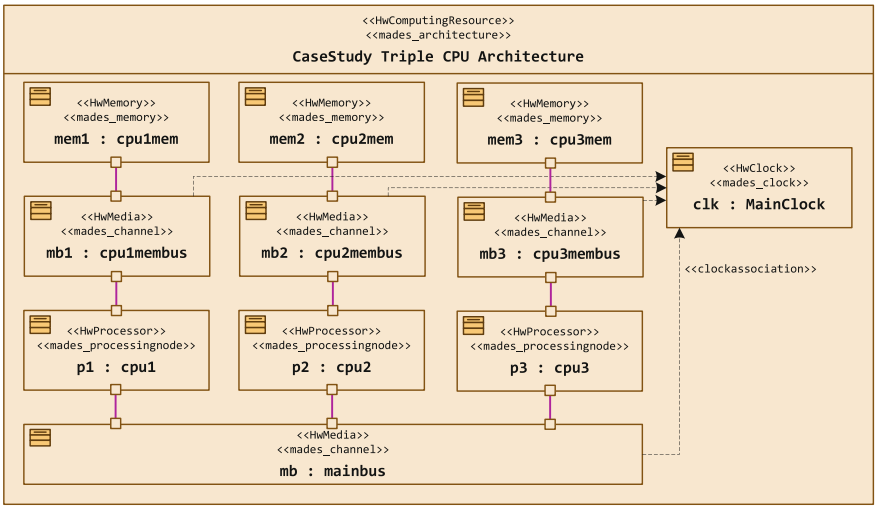


Fig. 17 Revised hardware specification diagram for the case study target architecture

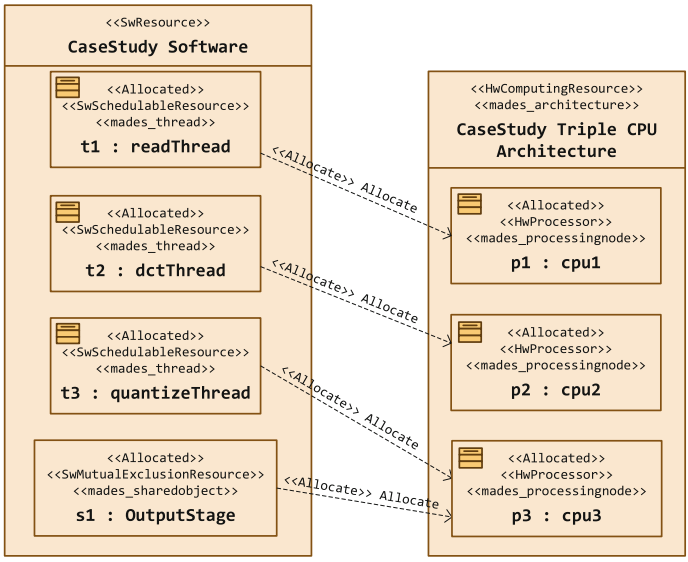


Fig. 18 Revised allocation diagram for the case study

7 Conclusions

This chapter has presented some of the major problems encountered when developing complex embedded systems. The hardware architectures of such systems are characterised by the use of non-standard, application-specific features, such as mul-

tiple heterogeneous processing units, non-uniform memory architectures, complex interconnect, on-chip networks, and custom function accelerators. These features are poorly supported by the programming languages most commonly used by industry for embedded development (such as C, C++ and Java) because these languages assume a 'standard' architecture with a simple programming model. Furthermore, many embedded systems are real-time or safety-critical systems and so are subject to many additional restrictions that affect the development process. Existing approaches to solve these problems tend to lack industrial support; either because they complicate certification through the use of new languages and tools; because they prevent the use of legacy code; or because they are not integrated well enough into existing development processes.

The chapter then described AnvilJ, a novel approach for the development of embedded Java. Unlike most virtualisation systems that operate primarily at runtime, AnvilJ operates primarily at compile-time and uses a restricted programming model based on a technique called Compile-Time Virtualisation. This restricted model allows AnvilJ to operate with vastly reduced runtime support that is predictable and bounded. In addition, whilst the CTV model imposes restrictions on the programmer, these are shown to be less than is imposed by most real-time development processes.

In order to aid industrial acceptance, AnvilJ is integrated into a model-based engineering tool flow as part of the MADES project using traceable model transformations implemented in the Epsilon framework. MADES' modelling language is augmented with a small set of stereotypes to provide the additional modelling information required. The use of these transformations allows AnvilJ to be used by modellers and designers without manual intervention.

The use of model-driven engineering also allows the presented approach to automate the process of hardware development. An approach is shown which translates the hardware diagrams from the system model into VHDL, a hardware description language suitable for implementation on FPGAs. Whilst this does not expose the full flexibility of VHDL or the chosen implementation fabric, it can be used for rapid prototyping, functional verification, and design-space exploration. Due to the fact that the hardware generation transformation and the software generation transformation are described by the same metamodel, the generated software will execute correctly on the generated hardware.

To demonstrate the approach, the chapter showed a case study based on the vision subsystem of an automotive safety system. The required models are developed and passed to AnvilJ, which refactors the input code to target two different complex architectures without any code writing.

The use of AnvilJ does not make an unpredictable system predictable, however when used in an otherwise real-time development process it will not make the system less predictable. In general, worst-case execution time (WCET) analysis for complex embedded architectures is a significant open problem. Almost all of the schedulability and WCET analysis performed for uniprocessor systems no longer applies to multiprocessor systems and many worst-case analytical models of complex embedded hardware are still too pessimistic for real-world use. These issues

are being considered within the T-CREST [42] project which aims to build a time predictable NoC based multiprocessor architecture, with supporting compiler and WCET analysis.

References

1. J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill, Kaffemik—A distributed JVM on a single address space architecture. SCI Europe 2001 Conference (2001)
2. Atego. Perc Pico (2011), http://www.atego.com/products/aonix-perc-pico/
3. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In CODES '02 (2002), pp. 73–78
4. J. Bezivin. In Search of a Basic Principle for Model-Driven Engineering. UPGRADE–Eur. J. Inf. Prof. (2004)
5. A. Burns, B. Dobbing, G. Romanski, The ravenscar tasking profile for high integrity real-time programs. In Ada-Europe '98 (Springer, Berlin, 1998) pp. 263–275
6. A. Burns, A.J. Wellings, *Real-time systems and their programming languages* (Addison-Wesley Longman Publishing Co., Inc., Boston, 1990)
7. K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. IBM Syst. J. 45(3), (2006)
8. EE Times. Under the Hood–Update: Apple iPhone 3G exposed, December (2008), http://www.eetimes.com/design/microwave-rf-design/4018424/Under-the-Hood-Update-Apple-iPhone-3G-exposed
9. J. Eker, J. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, Taming heterogeneity—the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
10. D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, *SpecC: Specification language and design methodology* (Kluwer Academic Publishers, Boston, 2000)
11. R. Gonzalez, Xtensa: A configurable and extensible processor. Micro, IEEE **20**(2), 60–70 (2000)
12. J. Gosling, G. Bollella, *The real-time specification for java* (Addison-Wesley Longman Publishing Co., Inc., Boston, 2000)
13. I. Gray, N.C. Audsley, Developing predictable real-time embedded systems using AnvilJ. In Proceedings of The 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012) Beijing, China, April 17–19 (2012)
14. I. Gray, N. Audsley, Exposing non-standard architectures to embedded software using compile-time virtualisation. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09) 2009
15. I. Gray, N. Audsley, Supporting Islands of Coherency for highly-parallel embedded architectures using Compile-Time Virtualisation. In 13th International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2010
16. N. Guan, M. Lv, W. Yi, G. Yu, WCET analysis with MRU caches: Challenging LRU for predictability. In Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications, Symposium (RTAS) 2012
17. S. Gupta, N. Dutt, R. Gupta, A. Nicolau, SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In Proceedings of 16th International Conference on, VLSI Design, pp. 461–466, Jan. 2003
18. J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical assessment of MDE in industry. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, pp. 471–480 (2011)
19. Terracotta Inc., *The Definitive Guide to Terracotta–Cluster the JVM for Spring, Hibernate and POJO Scalability* (Apress, New York, 2008)

20. ITRS, International Technology Roadmap for Semiconductors, 2007 edn. (2007), http://www.itrs.net/
21. F. Jouault, J. Bézivin, M. Barbero, *Towards an advanced model-driven engineering toolbox* (Innovations Syst. Softw, Eng, 2009)
22. R. Klefstad, M. Deshpande, C. O'Ryan, A. Corsaro, A.S. Krishna, S. Rao, K. Raman, *The performance of ZEN: A real time CORBA ORB using real time Java* (In Proceedings of Real-time and Embedded Distributed Object Computing Workshop, OMG, Sept, 2002)
23. L.R.R.F.P.D.S. Kolovos, Extensible platform for specification of integrated languages for model management (Epsilon) (2010), http://www.eclipse.org/gmt/epsilon
24. J. Kwon, A. Wellings, S. King, Ravenscar-Java: A high integrity profile for real-time Java. In Joint ACM Java Grande/ISCOPE Conference (ACM Press, New York, 2002), pp. 131–140
25. P. Marwedel, *Embedded System Design* (Springer, New York, 2006)
26. T. Mattson, R.V. der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, *The 48-core SCC processor: the programmer's view* (Storage and Analysis (SC), In International Conference for High Performance Computing, Networking, 2010), p. 2010
27. Mentor Graphics. Catapult-C synthesis (2009), http://www.mentor.com/catapult
28. Modeliosoft. Modelio—The open source modeling environment (2012), http://www.modeliosoft.org/
29. P. Mohagheghi, V. Dehlen, Where Is the Proof?—A review of experiences from applying MDE, in *industry, In Model Driven Architecture U Foundations and Applications, vol. 5095, Lecture Notes in Computer Science*, ed. by I. Schieferdecker, A. Hartman (Springer, Berlin, 2008), pp. 432–443
30. Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (2009), http://www.omgmarte.org/
31. F. Pizlo, L. Ziarek, and J. Vitek, Real time Java on resource-constrained platforms with Fiji VM. In Proceedings of JTRES, JTRES '09, ACM, New York, pp. 110–119 (2009)
32. A.L. Pope, *The CORBA reference guide: understanding the Common Object Request Broker Architecture* (Addison-Wesley Longman Publishing Co., Inc., Boston, 1998)
33. J. Reineke, D. Grund, C. Berg, R. Wilhelm, Timing predictability of cache replacement policies. Real-Time Syst. **37**, 99–122 (2007). doi:10.1007/s11241-007-9032-3
34. M. Rivas, M. González Harbour, MaRTE OS: An Ada Kernel for real-time embedded applications, in *Reliable SoftwareTechnologies U Ada-Europe 2001*, vol. 2043, ed. by D. Craeynest, A. Strohmeier (Springer, Berlin, 2001), pp. 305–316
35. L.M. Rose, R.F. Paige, D.S. Kolovos, F.A. Polack. The Epsilon generation language. In ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture (Springer, Berlin, 2008), pp. 1–16
36. J.C.H. Roth, *Digital systems design using VHDL* (Pws Pub. Co., Boston, 1998)
37. D.C. Schmidt, D.L. Levine, S. Mungee, The design of the TAO real-time object request broker. Comput. Commun. **21**(4), 294–324 (1998)
38. Texas Instruments Inc. OMAP5430 mobile applications platform (2011), http://focus.ti.com/pdfs/wtbu/OMAP5_2011-7-13.pdf
39. The Eclipse Foundation. Eclipse Java development tools (2011), http://www.eclipse.org/jdt/
40. The MADES Consortium. The MADES Project (2011), http://www.mades-project.org/
41. The Motor Industry Software Reliability Association. Guidelines for the Use of the C Language in Critical Systems. MISRA Ltd., 2004
42. The T-CREST Consortium. The T-CREST Project (2012), http://www.3sei.com/t-crest/
43. T. Weilkiens, *Systems engineering with SysML/UML: Modeling, analysis design* (Morgan Kaufmann Publishers Inc., San Francisco, 2008)
44. D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, A. Agarwal, On-chip interconnection architecture of the tile processor. Micro, IEEE **27**, 15–31 (2007)
45. D. Wiklund, D. Liu, SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems. In IPDPS '03, p. 78.1 (2003)

46. Xilinx Corporation. Embedded System Tools Reference Guide—EDK 11.3.1. Xilinx Application, Notes, UG111 (2009)
47. Xilinx Corporation. Platform Studio and the Embedded Development Kit (EDK) (2012), http://www.xilinx.com/tools/platform.htm
48. Xilinx Corporation. Virtex-5 FPGA Configuration User Guide. Xilinx User Guides, UG191 (2006)

Part II
Design Patterns and Development Methodology

MADES EU FP7 Project: Model-Driven Methodology for Real Time Embedded Systems

Imran R. Quadri, Alessandra Bagnato and Andrey Sadovykh

Abstract The chapter presents the EU funded FP7 MADES project that focus on real-time embedded systems development. The project proposes a high abstraction level based model-driven methodology to evolve current practices for real-time embedded systems development in avionics and surveillance industries. In MADES, an effective SysML/MARTE language subset along with a set of new tools and technologies have been developed that support high-level design specifications, verification and automatic code generation, while integrating aspects such as component based *Intellectual Property* (IP) re-use. In this book chapter, we first present the MADES methodology and related diagrams developed to fulfill our goals; followed by a description of the underlying tool set developed in the scope of the MADES project. Afterwards, we illustrate the MADES methodology in the context of a car collision avoidance system case study to validate our design flow.

1 Introduction

Real-Time Embedded Systems (RTES) are gradually becoming an essential aspect of our professional and personal lives. From avionics, transport, defense, medical and telecommunication systems to general commercial appliances such as smart phones, high definition TVs, gaming consoles; these systems are now omnipresent, and it is difficult to find a domain where they have not made their mark. However, rapid

I. R. Quadri · A. Sadovykh
Softeam, 21 Avenue Victor Hugo, Paris 75016, France
e-mail: imran.quadri@softeam.fr

A. Sadovykh
e-mail: andrey.sadovykh@softeam.fr

A. Bagnato (✉)
TXT e-solutions, Via Frigia 27, 20126 Milan, Italy
e-mail: alessandra.bagnato@txtgroup.com

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_3, © Springer-Verlag Berlin Heidelberg 2014

evolution and continuous technological advances in the underlying hardware/software along with sharp increase in targeted application domains have lead to new challenges in the design specification and implementation of RTES, such as increasing costs, increase in time to market and augmentation in the design gap between hardware and software evolution. Currently, we are therefore faced with a need to design more effective RTES. Hence effective design methodologies are needed to decrease the overall development costs, while resolving issues such as related to system complexity, verification and validation, etc.

Various methodologies and propositions have been proposed for this purpose. A *Platform* or *component based approach* is widely accepted in the RTES industry, permitting system conception and eventual design in a compositional manner. The hierarchy related to the RTES is visible quite clearly, and designers are capable to re-use components that have been either developed internally or by third parties. Other methodologies make use of high abstraction levels, in order to elevate the low level technical details. In addition, these systems should also be eventually developed, and efforts must be made to maximize debugging and testing for minimizing the manufacturing costs, power consumption levels and system size.

It is in the context of improving the primary productivity of RTES, that this chapter finds its proper place. One of the primary guidelines followed during these works is the utilization of Model-Driven Engineering (MDE) [1] for RTES specification and development. MDE is able to benefit from a component based model-driven approach, allowing to abstract and simplify the system specifications using UML (Unified Modeling Language) graphical language [2], while enabling the possibility of integrating a compilation chain to transform the high-level models to executable code for eventual implementation in execution platforms, such as Application-Specific Integrated Circuits (ASICs) or Field-Programmable Gate Arrays (FPGAs).

MDE enables to elevate as well as partition the system design: by enabling parallel independent specifications of both system hardware and software; their eventual allocation, and the possibility of integrating heterogeneous components into the system. Usage of UML increases system comprehensibility as it enables designers to provide high-level descriptions of the system, easily illustrating the internal concepts (system hierarchy, flows/connections, control/data dependencies etc.). The graphical nature of these specifications equally enables reuse or refinements, depending upon underlying tools and user requirements. Additionally, MDE englobes different technologies and tools such as UML *profiles* for high-level system specifications and *model transformations* [3]. These transformations can automatically generate executable models or code from the abstract high-level design models.

The contributions of this chapter relate to presenting a complete methodology for the design of RTES in the scope of the EU funded FP7 MADES [4, 5] project. MADES aims to develop novel model-driven techniques to improve existing practices in development of RTES for avionics and surveillance embedded systems industries. It proposes an effective subset of existing standardized UML profiles for embedded systems modeling: SysML [6] and modeling and analysis of real-time and embedded systems (MARTE) [7], while avoiding incompatibilities resulting from simultaneous usage of both profiles. The MADES methodology integrates new

tools and technologies that support high-level SysML/MARTE system design specifications, their verification and validation (V&V), component re-use, followed by automatic code generation to enable execution platform implementation.

The contribution related to presenting the MADES methodology based on mixed SysML/MARTE usage is of utmost importance. While a large number of works deal with embedded systems specifications using only either SysML or MARTE, we present a combined approach and illustrate the advantages of using these two profiles. This aspect is significant in nature as while both these profiles provide numerous concepts and supporting tools, they are in turn difficult to be mastered by system designer. For this purpose, we present the MADES language, which focuses on an effective subset of SysML and MARTE profiles and proposes a specific set of unique diagrams for expressing different aspects related to a system, such as hardware/software specifications and their eventual mapping. In the paper, an overview of the MADES language and the associated diagrams is presented, that enables rapid design and incremental composition of system specifications. The resulting models then can be taken by the underlying MADES tool set for goals such as component re-use, verification or automatic code generation, which are also briefly detailed in the chapter.

Afterwards, we illustrate the various concepts present in the MADES language by means of an effective real-life embedded systems case study: a car collision avoidance system (CCAS) that integrates the MADES language and illustrates the different phases of our design methodology and implementation approach. This case study serves as a reference guide to the actual case studies provided by the MADES end users: more specifically an onboard radar control unit provided by TXT e-solutions and a ground based radar processing unit provided by Cassidian (an EADS company). Hence, the results obtained from the CCAS case study are in turn integrated in the actual MADES case studies.

2 Background: Using SysML and MARTE for RTES Design and Development

In this section, we first provide an overview of the SysML and MARTE profiles and then describe the related works focusing on their usage. While a large number of researches exist that make use of either SysML or MARTE for high-level modeling of RTES, due to space limitations, it is not possible here to give an exhaustive description and we only provide a brief summary on some of the works that make use of SysML or MARTE based high abstraction levels and MDE, for RTES design specification and implementation.

2.1 Systems Modeling Language

System Modeling Language (SysML) is the first UML standard for system engineering proposed by Object Management Group [8] that aims at describing complex systems. SysML allows describing of the functional requirements in graphical or tabular form to aid with model traceability, and provides means to express the composition of the system by means of *blocks* and related behavior by means of UML inspired *Activities*, *Interactions*, *State Machines*, etc. This profile also provides the designer with parametric formalisms which are used to express analytical models based on equations.

However, while SysML is used in the RTES community, it was not mainly created for modeling of embedded system designs. Non-functional properties such as timing constraints, latency and throughput that are crucial for the design of RTES are absent in this profile. This is not the case of the UML MARTE profile.

2.2 Modeling and Analysis of Real-Time and Embedded Systems

The MARTE profile extends the possibilities to model the features of software and hardware parts of a real-time embedded system and their relations. It also offers added extensions, for example to carry out performance and scheduling analysis, while taking into consideration the platform services (such as the services offered by an OS). The profile is structured in two directions: first, the modeling of concepts of real-time and embedded systems and secondly, the annotation of the models for supporting analysis of the system properties. These two major parts share common concepts: for expressing non-functional properties (*NFPs*), timing notions, resource modeling (such as computing, storage resources), UML inspired components based modeling (concepts such as classes, instances, port and connectors) and allocation concepts, among others.

Additionally, MARTE contains certain concepts present in other standards and frameworks, which permit to increase synergy between designers of different domains. Architecture Analysis and Design Language (AADL), that has its origins in the avionic domain, is a SAE[1] standard for the development of real-time embedded systems. In [9], the authors compared the relationship between AADL and MARTE concepts. Similarly, Automotive Open System Architecture (AUTOSAR) [10] is a standardized and open automotive software architecture framework, developed jointly by different automobile manufacturers, suppliers and tool developers. With regards to AUTOSAR, MARTE already covers many aspects of timing: such as specification of *over-sampling* and *under-sampling* in end-to-end timing chains (commonly found in complex control systems). In [11], the SPIRIT consortium's IP-XACT UML profile has been proposed, which is a specialization of the current MARTE profile.

[1] Society of Automotive Engineers: http://www.sae.org/servlets/index.

2.3 Related Works

The MoPCoM project [12] uses MARTE profile to target modeling and code generation of reconfigurable embedded systems. While the project inspires from SysML concepts such as requirements and blocks, they are not fully integrated in the design flow. The project uses the IBM Harmony[2] process coupled with Rhapsody[3] UML modeling tool. Additionally, MoPCoM proposes two distinct flows for system modeling and schedulability analysis that increase design efforts. Similarly, eDIANA [13] is an ARTEMIS project that uses MARTE profile for RTES specification and validation. However, detailed specification of software and hardware aspects are not illustrated in the project. While TOPCASED [14] differs from MADES, as it focuses primarily on IDE infrastructure for real-time embedded systems and not on any particular implementations.

Project SATURN [15] is another EU FP7 project that aims to use high-level co-modeling approach for RTES simulation and synthesis goals. However, the project only takes SysML into account and proposes multiple UML profiles, for co-simulation, synthesis and code generation purposes, respectively. The goal is to use carry out hardware/software modeling via these profiles and generate SystemC for eventual VHDL translation and FPGA implementation. Unfortunately, the project does not utilizes the MARTE standard for hardware/software Co-Design [16] modeling and increases the learning curve due to the introduction of several new dedicated profiles.

In [17], the authors provide a mixed modeling approach based on SysML and the MARTE profiles to address design space exploration strategies. However, the shortcomings of this approach is that they only provide implementation results by means of mathematical expressions and no actual experimental results were illustrated. The OMEGA European project [18] is also dedicated to the development of critical real-time systems. However it uses pure UML specifications for system modeling and proposes a UML profile [19], which is a subset of an earlier UML profile for Scheduling, Performance and Time (SPT) [20], that has been integrated in MARTE. The MARTES project emphasizes on combined usage of UML and SystemC for systematic model-based development of RTES. The results from this project in turn, have contributed to the creation of the MARTE profile. While the EU FP7 INTERESTED project [21] proposes a merged SysML/MARTE methodology where SysML is used for requirement specifications and MARTE for timing aspects, unfortunately it does not proposes rules on combined usage of both profiles.

The MADES project aims to resolve this issue and thus differentiates from the above mentioned related works, as it focuses on an effective language subset combining both SysML and MARTE profiles for rapid design and specification of RTES. The two profiles have been chosen as they are both widely used in embedded systems design, and are complimentary in nature [22]. MADES proposes automatic generation of hardware descriptions and embedded software from high-level models, and

[2] http://www-01.ibm.com/software/rational/services/harmony/

[3] http://www-01.ibm.com/software/awdtools/rhapsody/

integrates verification of functional and non-functional properties, as illustrated in the subsequent section.

Thus as evident from the previously cited related works, both SysML and MARTE are being widely used in both the academia as well as the real-time embedded systems industry. SysML is normally used for high-level system design specifications and requirement engineering, while MARTE enables the possibility to enrich a system specification with non-functional properties, hardware/software Co-Design along with timing, performance and schedulability analysis aspects. A merge of both SysML and MARTE thus seems quite logical, as it enables a designer to carry out SysML based high-level requirements and functional system descriptions and then enrich these models with MARTE concepts.

3 MADES Model-Driven Design Methodology

In this section, we provide a brief overview of the MADES design methodology, as illustrated in Fig. 1. Initially, the high-level system design models are carried out using the MADES language and associated diagrams, which are represented later on in Sect. 3.1. After specification of the design models that include user requirements, related hardware/software aspects and their eventual allocation; underlying

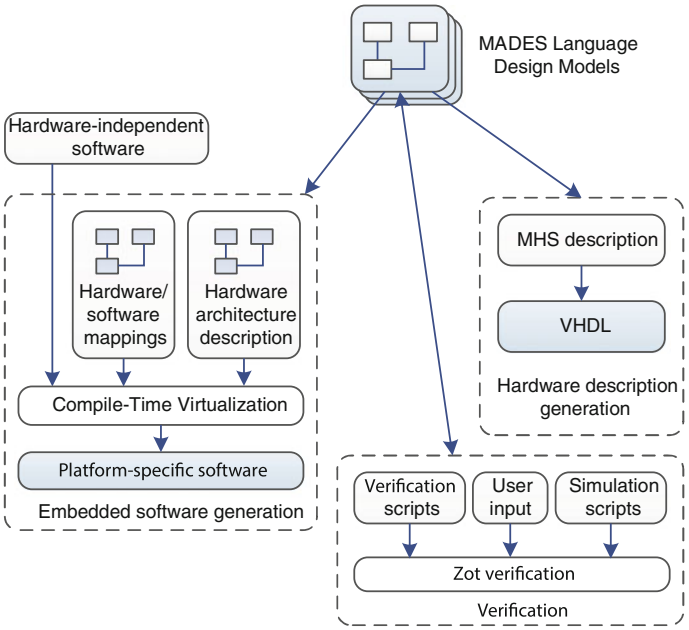


Fig. 1 An overview of the MADES methodology

model transformations (*model-to-model* and *model-to-text* transformations) are used to bridge the gap between these abstract design models and subsequent design phases, such as verification, hardware descriptions of modeled targeted architecture and generation of platform-specific embedded software from architecturally neutral software specifications. For implementing model transformations, MADES uses the Epsilon platform [23], that enables model transformations, code generation, model comparison, merging, refactoring and validation [24].

Verification activities in MADES include verification of key properties of designed concepts (such as meeting deadlines, etc.) and that of model transformations integrated in the design flow [25, 26]. For verification and simulation purposes, MADES uses the Zot tool [27], which permits the verification of, among others, aspects such as meeting of critical deadlines. While closed-loop simulation on design models enables functional testing and early validation.

Additionally, MADES employs the technique of *Compile-Time Virtualization* (CTV) [28], for targeting of non-standard hardware architectures, without requiring development of new languages or compilers. Thus a programmer can write architecturally neutral code which is automatically distributed by CTV over a complex target architecture. Finally, via model transformations, code generation (for example; such as either in VHDL for hardware, and Real-Time Java for software) can be carried that can be eventually implemented on modern state-of-the-art FPGAs. Currently MADES model transformations target Xilinx FPGAs, however it is also possible for them to adapt to FPGAs provided by other vendors such as Altera or Atmel. A detailed description regarding the global MADES methodology can be found in [29, 30].

3.1 MADES Language and Related Diagrams

Figure 2 gives an overview of the underlying MADES language present in the overall methodology for the initial model based design specifications. The MADES language focuses on a subset of SysML and MARTE profiles and proposes a specific set of diagrams for specifying different aspects related to a system: such as requirements, hardware/software concepts, etc. Along with these specific diagrams, MADES also uses classic UML diagrams such as *State* and *Activity* diagrams to model internal behavior of system components, along with *Sequence* and *Interaction Overview* diagrams to model interactions and cooperation between different system elements. Softeam's Modelio UML Editor and MDE Workbench [31] enables full support of MADES diagrams and associated language, as explained later on in Sect. 4.1. We now provide a brief description of the MADES language and its related diagrams.

In the initial specification phase, a designer needs to carry out system design at high abstraction levels. This design phase consists of the following steps:

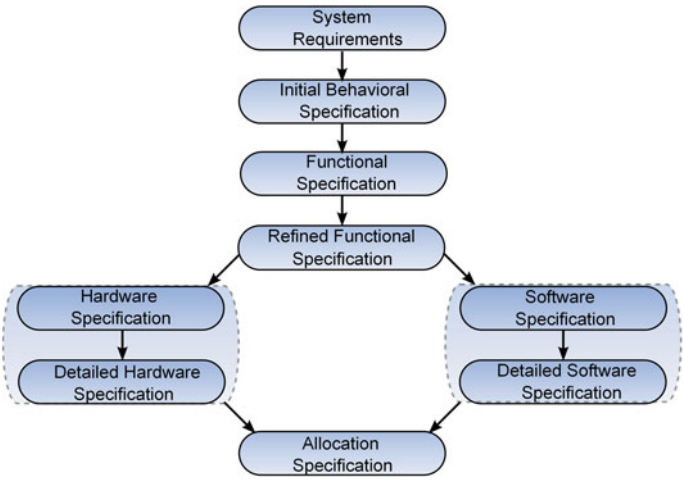


Fig. 2 Overview of MADES language design phases

- *System Requirements*: The user initially specifies the functional requirements related to the system. For this purpose, a MADES *Requirements Diagram* is utilized that integrates SysML requirements concepts.
- *Initial Behavioral Specification*: Afterwards, initial behavioral specification is carried out by means of UML use cases, interactions, state machines or activities during the preliminary analysis phase.
- *Functional Specification*: Once the behavioral specifications are completed, they are then linked to SysML blocks (or internal blocks) by means of MADES *Functional Block* (or *Internal Functional Block*) *Specification Diagram*, that contains SysML block (or internal block) concepts. This functionality is independent of any underlying execution platform and software details. It thus determines 'what' is to be implemented, instead of 'how' it is to be carried out.
- *Refined Functional Specification*: This level refines SysML aspects into MARTE concepts: The *Refined Functional Specification Diagram* models MARTE components, each corresponding to a SysML block. Here, MARTE's *High level Application Modeling* package is used to differentiate between active and passive components of the system.

The refined functional specification phase links SysML and MARTE concepts but avoids conflicts arising due to parallel usage of both profiles [22]. Only the SysML and MARTE allocation aspects available in both profiles are used in the refined functional specification design phase to avoid any possible conflict. While the allocation concept is present both in SysML and MARTE, MARTE enriches the basic SysML allocation aspects and is thus the one adopted for our methodology. SysML is used for initial requirements and functional description, while MARTE is utilized for the enriched modeling of the global functionality and execution platform/software modeling along with their allocations, creating a clear separation between the two

profiles. Afterwards, the designer can move onto the hardware/software partitioning of the refined functional specifications. These following steps are elaborated by means of MARTE concepts.

Related to the MARTE modeling, an allocation between functional and refined functional level specifications is carried out using a MADES *Allocation Diagram*. Afterwards, a Co-Design approach is used to model the hardware and software aspects of the system. The modeling is combined with MARTE *Non-Functional Properties* and *Timed Modeling* package to express aspects such as throughput, temporal constraints, etc. We now describe the hardware and software modeling, which are as follows:

- *Hardware Specification*: The MADES *Hardware Specification Diagram* in combination with concepts defined in MARTE's *Generic Resource Modeling* package enables modeling of abstract hardware concepts such as computing, communication and storage resources. This phase enables a designer to describe the physical system in a generic manner, without going into too much details regarding the implementation aspects. By making use of MARTE GRM concepts, a designer can describe a system such as a car, a transport system, flight management system, among others.
- *Detailed Hardware Specification*: Using the *Detailed Hardware Specification Diagram* with MARTE's *Hardware Resource Modeling* package allows extension, refinement or enrichment of concepts modeled at the hardware specification level. It also permits modelling of systems such as FPGA based System-on-Chips (SoCs), ASICs etc. A one-to-one correspondence usually follows here: for example, a computing resource typed as MARTE `ComputingResource` is converted into a hardware processor, such as a PowerPC or MicroBlaze [32], effectively stereotyped as MARTE `HwProcessor`. Afterwards, an *Allocation Diagram* is then utilized to map the modeled hardware concepts to detailed hardware ones.
- *Software Specification*: The MADES *Software Specification Diagram* along with MARTE's *Generic Resource Modeling* package permits modeling of software aspects of an execution platform such as schedulers and tasks; as well as their attributes and policies (e.g. priorities, possibility of preemption).
- *Detailed Software Specification*: The MADES *Detailed Software Specification Diagram* and related MARTE's *Software Resource Modeling* are used to express detailed aspects of the software such as an underlying *Operating System*, (OS), threads, address space, etc. Once this model is completed, an *Allocation Diagram* is used to map the modeled software concepts to detailed software ones: for example, allocation of tasks onto OS processes and threads. This level can express standardized or designer based RTOS APIs. Thus multi-tasking libraries and multi-tasking framework APIs can be described here.
- *Clock Specification*: The MADES *Clock Specification Diagram* (not shown in Fig. 2) is used to express timing and clock constraints/aspects. It can be used to specify the physical/logical clocks present in the system and the related constraints. This diagram makes use of MARTE's *Time Modeling* concepts such as clock types

and related constraints. Here, designers model all the timing and clock constraint aspects that could be used in all the other different phases.

Iteratively, several allocations can be carried out in our design methodology: an initial software to hardware allocation may allow associating schedulers and schedulable resources to related computing resources in the execution platform, once the initial abstract hardware/software models are completed, in order to reduce *Design Space Exploration* (DSE).

Subsequently this initial allocation can be concretized by further mapping of the detailed software and hardware models (an allocation of OS to a hardware memory, for example), to fulfill designer requirements and underlying tools analysis results. An allocation can also specify if the execution of a software resource onto a hardware module is carried out in a sequential or parallel manner. Interestingly, each MADES diagram only contains commands related to that particular design phase, thus avoiding ambiguities of utilization of the various concepts present in both SysML and MARTE, while helping designers to focus on their relative expertise. Additionally, UML behavioral diagrams in combination with MADES concepts (such as those related to verification) can be used for describing detailed behavior of system components or the system itself.

Finally, the MADES language also contains additional concepts used for the underlying model transformations for code generation and verification purposes, which are not present in either SysML or MARTE, and are detailed in [29]. Once the modeling aspects are completed, verification and code generation can be carried out. These aspects are out of the scope of this chapter, and we refer the reader to [29, 33] for complete details.

4 MADES Tool Set

We now describe the MADES tool set that enables to move from high level SysML/MARTE modeling to verification, code generation and eventual implementation in execution platforms.

4.1 Modelio UML Editor and MDE Workbench

In the frame of the MADES project, Softeam [34] has developed a dedicated extension to its Modelio UML Editor and MDE Workbench. Modelio fully supports the MADES methodology and underlying language while providing various additional features such as automatic document generation and code generation for various platforms. Modelio is highly extensible and can be used as a platform for building new MDE features. The tool allows building UML2 Profiles, combined with a reach graphical interface for dedicated diagrams, model element properties editors and

action commands controls. The users have access to several extensions mechanisms: light Python scripts or Java API. Finally, Modelio is available in both open source and commercial versions, and nearly all the MADES diagrams are present in the open source version of Modelio (all except the SysML inspired requirements specifications), in order to carry out RTES modeling, using the MADES methodology.

As seen in Fig. 3, Modelio has developed unique MADES diagrams, as specified earlier in Sect. 3.1, along with a set of unique commands for each specific diagram and related design phase. Thus, when a designer is working on a particular phase that suits his/her particular expertise: such as detailed hardware specification, he/she will be able to create concepts such as processors, RAM/ROM memories, caches, bridges, buses etc. The advantage of this approach is that designers do not have to understand the various concepts present in SysML and MARTE and do not need to guess which UML concept (such as classes, instances, ports) should be applicable to which particular design phase, and which particular stereotype should be applied to that concept. The commands are also assigned simple names in order for someone not highly familiar with SysML/MARTE standards to guess at their functionality. For example, in the figure, the `Processor` command present in the `Class model` section of the command palette signifies the possibility of creation of a processor class. This command in turn automatically creates a dually stereotyped class with two stereotypes: a MARTE `HwProcessor` stereotype and a MADES stereotype `mades_processingnode`. The second stereotype is used by the underlying model transformations for verification and code generation purposes, in a complete transparent manner for the end user. It should be observed that these additional concepts which are not present in either SysML/MARTE are not needed to be mastered by a designer carrying out the model based specifications. A designer just needs to determine which concepts from the palette will be needed for modeling of the

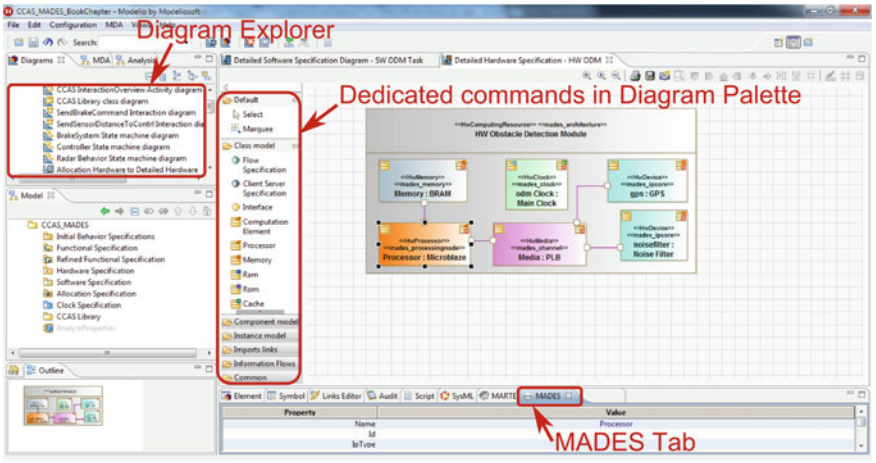


Fig. 3 A screenshot of Modelio illustrating the MADES diagrams/command set

platform and the underlying MADES concepts are added automatically to these concepts, thanks to a mapping between SysML/MARTE concepts and those needed by the model transformations. This mapping has been defined in [29] and is only needed for the detailed software/hardware specification design phases. Finally, overall design time and productivity can be increased due to the development of specific diagram set and related commands in Modelio, available both in the open source and commercial versions.

4.2 MADES Component Repository

The MADES Component Repository (CRP), as shown in Fig. 4, is used to store, search and download MADES components created by the MADES developers with Modelio. The Component Repository module accesses a central MADES component database while offering various web services to manage uploading, searching or downloading of the components stored within the database, and the queries that have been performed on its contents. The offered web services are accessible through an ad-hoc Component Repository web-based flexible graphical user interface, as seen in Fig. 5 or directly through Modelio itself.

Thus the CRP enables *Intellectual Property* (IP) re-use, enabling designers to create, store or re-use IP blocks to build different applications, platforms or complete systems, while reducing design time. Complete details about the MADES CRP can be found in [29, 33].

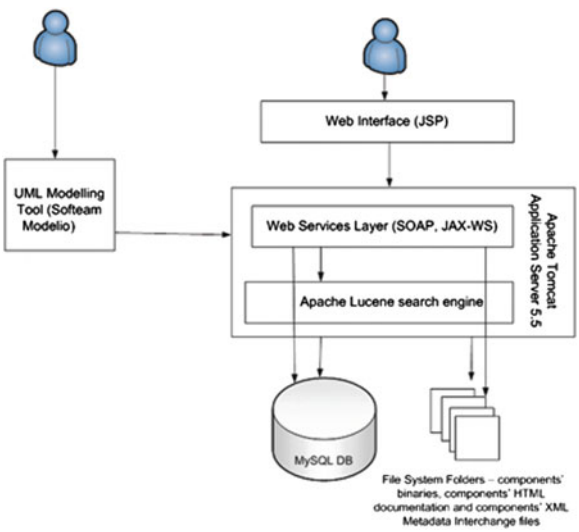


Fig. 4 Block structure of the MADES component repository

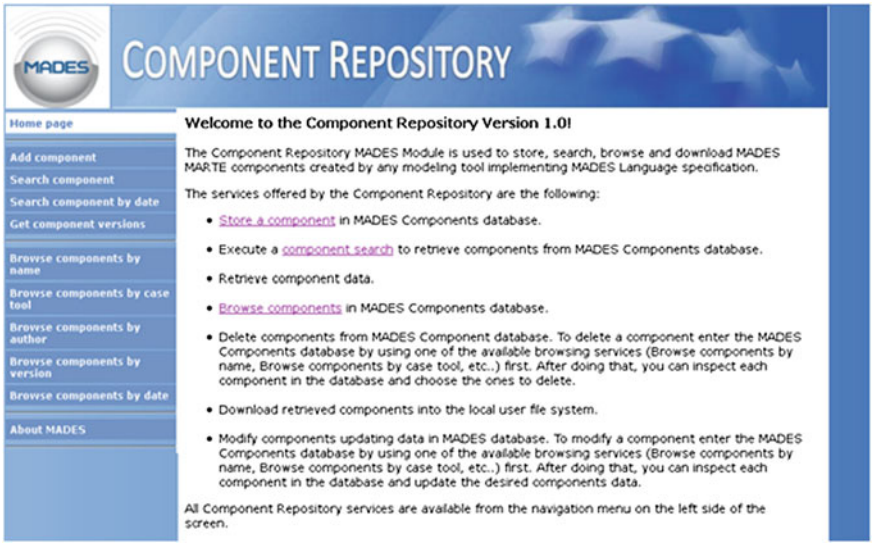


Fig. 5 The MADES component repository welcome page

4.3 Zot Verification Tool

Verification is carried out by transforming MADES diagrams into temporal logic formulae, using the semantics defined in [26]. These are, in turn, fed to the Zot verification tool, which signals whether the stated property holds for the modeled system or not, and in the latter case, returns a counterexample, i.e., a system trace violating the property.

In fact, once the temporal logic model is created from the diagrams describing the system, the Zot tool can be used in two ways: to check whether user-defined properties hold for the system; and to produce traces compatible with a formal model, in what amounts to a simulation of the system. The simulation capabilities of the Zot tool can be used, as described in [35], in combination with a simulation tool such as OpenModelica [36] to perform closed-loop simulations of the designed embedded system with its physical environment.

4.4 MADES Model Transformations

The underlying MADES model transformations focus on several areas, such as generation of platform-specific software from architecturally-neutral software specifications using CTV. The model transformations are capable of transforming user-provided, hardware independent code and rewriting it to target the modeled hardware architecture. The transformation builds a minimal overhead runtime layer

to Implement the modeled system, and translates the user-provided software to make use of this layer. If the hardware or allocations are changed in the model then the generated runtime layer is automatically reduced or expanded accordingly.

Additionally, generation of hardware descriptions of the modeled target architecture is possible, as the MADES transformations allow for the generation of implementable hardware descriptions of the target architecture from the input system modeled via Modelio. The hardware related model transformations generate hardware description for input to standard commercial FPGA synthesis tools, such as Xilinx ISE and EDK tools. Presently, the model transformation are capable of generation *Microprocessor Hardware Specification* (MHS) which can be taken by Xilinx tools to generate the underlying hardware equivalent to that modeled using the MADES language.

The model transformations also enable verification of functional/non-functional properties, as results from Zot are fed back into Modelio in order to give the user feedback on the properties and locate errors, if any are found. The code generation facilities present in the model transformations are used to integrate the back-end of the verification tool, which Zot, with the front-end, which are the models expressed using the MADES language. Traceability support is also integrated in the model transformations for tracing the results of the verification activity back to the models, for tracing the generated code back to its source models and finally for tracing requirements to model elements such as use cases or operations, as well as to implementation files and test cases.

Thus these model transformations assist with mapping the programmer's code to complex hardware architectures, describing these architectures for implementation (possibly as an ASIC or on an FPGA) and verifying the correctness of the final system. Thus, while MADES does not support automatic hardware/software partitioning of a system, it enables designers to carry out automatic hardware/software generation of their specified models and enable software refactoring. Detailed descriptions about these model transformations, along with their installation and usage guidelines have been provided in [29, 33].

5 MADES Methodology in Practice-Car Collision Avoidance System Case Study

The car collision avoidance system or CCAS case study for short, when installed in a vehicle, detects and prevents collisions with incoming objects such as cars and pedestrians. The CCAS contains two types of detection modules. The first one is a radar detection module that emits continuous waves. If a transmitted wave collides with an incoming object, it is reflected and received by the radar itself. The radar sends this data to an obstacle detection module (ODM), which in turn removes the noise from the incoming signal along with other tasks such as a correlation algorithm.

The distance of the incoming object is then calculated and sent to the controller for appropriate actions.

The image processing module is the second detection module installed in the CCAS. It permits to determine the distance of the car from an object by means of image tracking. The camera takes pictures of incoming objects and sends the data to the image processing module, which executes a distance algorithm. If the results of the computation indicate that the object is closer to the car than a specified default value that means a collision can occur. The result of this data is then sent to the controller. The controller when receiving the data, acts accordingly to the situation at hand. In case of an imminent collision, it can carry out some emergency actions, such as stopping the engine, applying emergency brakes; otherwise if the collision is not imminent, it can decrease the speed of the car and can apply normal brakes.

The CCAS system development is described in detail subsequently. It should be mentioned that various modeled components present in the case study are also stored in the MADES CRP to serve as hardware/software *product catalogues*. For example, a component showcasing a radar functionality can be re-used in another modeled application dealing with an on board or ground based radar system. Similarly, a Discrete Cosine Transformation or DCT[4] algorithm inside the image tracking subsystem can have several implementations such as 1-D or 2-D based, which can be stored in the CRP with different version names. Depending upon end user requirements and Quality of Service criteria (performance, power consumption etc.), a designer can swap one implementation with the other, facilitating IP re-use.

The CCAS design specifications start with SysML based modeling, which involves the initial design decisions such as system requirements, behavioral analysis and functionality description, before moving onto MARTE based design phases (Fig. 6).

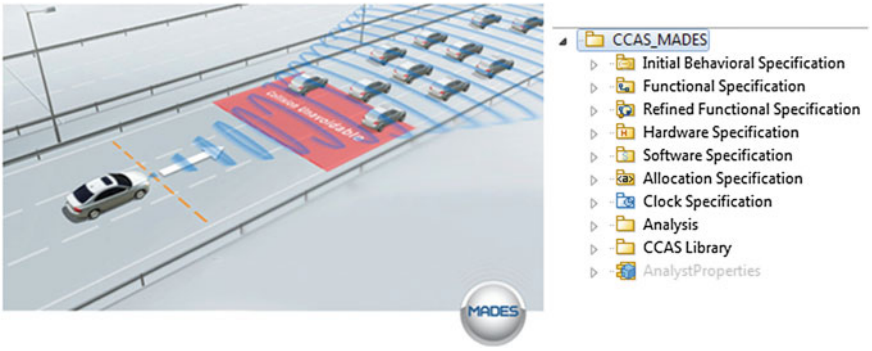


Fig. 6 The CCAS installed on a car to avoid collisions with incoming objects

[4] http://en.wikipedia.org/wiki/Discrete_cosine_transform

5.1 Requirements Specification

Using the SysML inspired MADES *Requirements Diagram*, system requirements are described at the initial system conception phase. Here in the particular case of CCAS, all the CCAS requirements were imported in Modelio from pre-existing Microsoft Excel files; and the gathered requirements were restructured, and can be enriched using SysML requirements (such as the requirements being traced, refined or satisfied), as seen later on in Sect. 5.4.

In Fig. 7, we illustrate the different requirements of the CCAS system. It should be mentioned that only the *functional* requirements of a system are described at this level. Here, the different requirements for the CCAS are described: the `Global Collision Avoidance Strategy` determines the global requirement for the system which is to detect incoming objects by means of either the radar or the image processing system. Additional requirements can be derived from this global requirement as shown in the figure shown above. It should be noted that this requirement specification has a strong relation with other MADES diagrams. More specifically, these specifications rely on initial behavioral and the functional specification phases, for their completion, as elaborated later on.

The CCAS requirements state that if the distance from an object is less than 3 m than the CCAS should enter in a warning state. If it remains in that state for 300 ms and distance is still less than 3 m, then the CCAS should decrease car speed and alert the driver by means of an alarm and a Heads-up-display (HUD). If the distance falls to 2 m, the CCAS should enter in a critical warning state. If it remains in that state for 300 ms and distance is still less than 2 m, then CCAS should apply emergency brakes, deploy airbags and alert the driver as well.

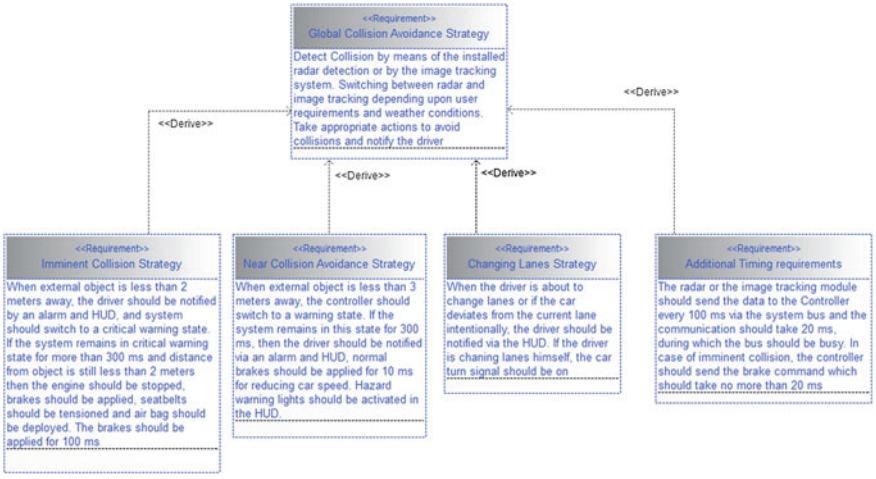


Fig. 7 The global system requirements related to the CCAS

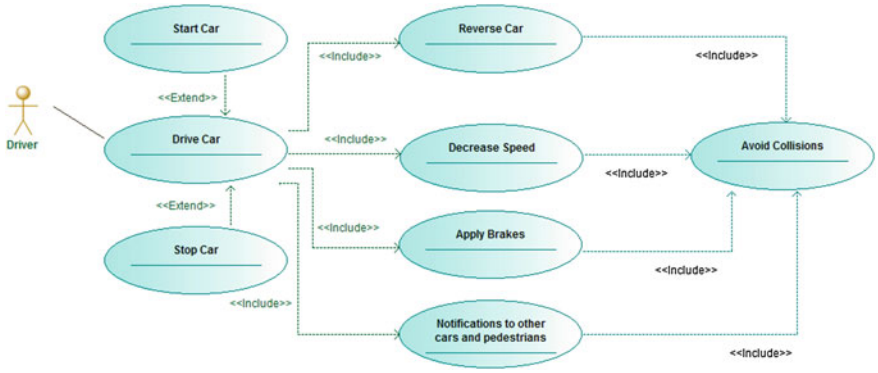


Fig. 8 The different case scenarios related to the CCAS

5.2 Initial Behavioral Specification

Once the requirement phase is partially completed, the next step is to describe the initial behavioral specifications associated with the system. For the particular case of CCAS, use cases are used to define the different scenarios associated with a car on which the CCAS is installed, as shown in Fig. 8. The creation of a MADES use case specification package guided the user by automatically creating a top level *Use Case Diagram* using built-in features in Modelio. The `Avoid Collisions` scenario makes use of other specified scenarios and is the one that is related to the system requirements described earlier.

5.3 Functional Specification

Once the requirements and use case scenarios of our system are specified; we move onto the functional block description of the CCAS system as described in Fig. 9. For this, MADES *Functional Block Specification* or *Internal Functional Block Specification Diagram(s)* are used. This conception phase enables a designer to describe the system functionality without going into details how the functionality is to be eventually implemented. Here the functional specification is described using SysML block definition diagram concepts. These functional blocks represent well-encapsulated components with thin interfaces that reflect an idealized system modular architecture. The functional description can be specified by means of UML concepts such as aggregation, inheritance, composition etc. Equally, hierarchical composition of functional blocks can be specified by means of internal blocks, ports and connectors. Here we use these concepts for describing the global composition of the CCAS. The `Car` block is composed of some system blocks such as a `Ignition System`, `Charging System`, `Starting System`,

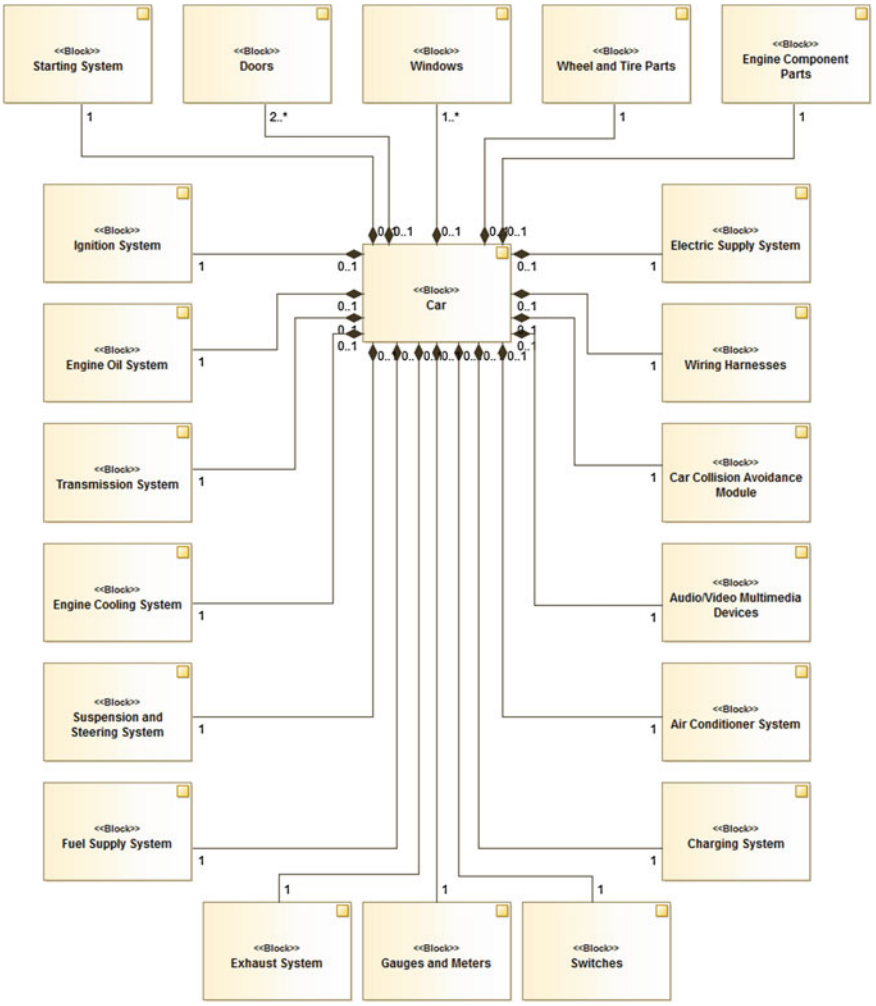


Fig. 9 Functional specification of the CCAS

`Engine Component Parts`, `Transmission System` and finally the `Car Collision Avoidance Module` which is the main component related to our case study. Each functional block can be composed of internal blocks, however, this step has not been illustrated in the chapter. Additionally, the initial behavioral specifications (Use cases in the case of CCAS) are mapped to appropriate functional blocks (in this particular case, the `Avoid Collisions` Use case is allocated onto the `Car Collision Avoidance Module` as seen in Fig. 10. This illustrates how the behavioral specifications are realized by the functional structure of the CCAS. In the MADES methodology, two types of allocations can be encountered; either for a refinement or for a Co-Design (software/hardware mapping). In this particular case,

Fig. 10 Mapping the `Avoid Collisions` use case to the `Car Collision Avoidance Module` block

the first type is used [29]. For the sake of simplicity, in the chapter, a refinement allocation (colored in orange) illustrates its related tagged values while these aspects are omitted for the Co-Design allocation (colored in red).

5.4 Completing the Requirements

Having completed the previous steps, it is now possible to complete the requirement specifications, as described in Fig. 11. As seen here, a related use case scenario and a functional block have been added to the figure, which helps to complete and satisfy the functional requirements. It should be noted that as seen in the figure, the `Car Collision Avoidance Module` block is utilized to satisfy the global system requirements, so it is this module that is the focus of the subsequent design phases.

Once the initial design descriptions have been specified, it is possible to partition and enrich the high-level functionalities. For this, MARTE concepts are used to

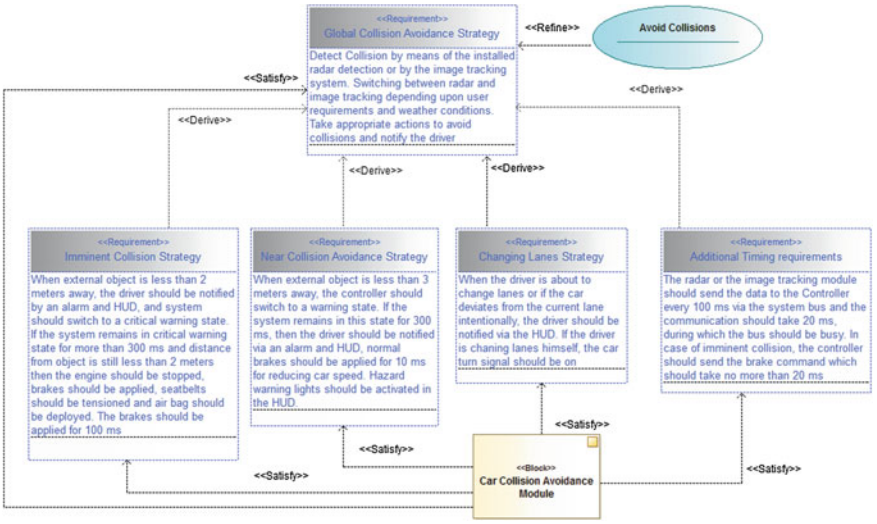


Fig. 11 Completing the functional requirements of the CCAS

determine which parts of the system are implemented in software or hardware along with their eventual allocation. Additionally, MARTE profile enables the expression of non-functional properties (NFP) related to a system, such as throughput, worst case execution times, etc. The subsequent MARTE based design phases are described in the following.

5.5 Refined Functional Level Specification

We now turn towards the MARTE based modeling of the CCAS. All necessary concepts present at the *Functional Level Specification Diagram* correspond to an equivalent (or refined) concept at the *Refined Functional Level Specification Diagram*. Since we are only interested in the `Car Collision Avoidance Module` at the functional level specification, an equivalent MARTE component is created. The `RH_Car Collision Avoidance Module` is stereotyped as a MARTE `RtUnit` that determines the active nature of the component. Figure 12 shows the related modeling of this concept. The `RtUnit` modeling element is the basic building block that permits to handle concurrency in RTES applications [7]. It should be mentioned that component structure and hierarchy should be preserved between the functional and refined functional level specification diagrams. As in this particular example, no hierarchical compositions are present at the functional level specifications for `Car Collision Avoidance Module`, they are equally not present in the underlying refined functional level specifications.

5.6 Allocating Functional and Refined Functional Level Specifications

Afterwards, a refinement allocation using the MADES *Allocation Diagram* is used to map the functional level specification concepts to the refined functional level specification concepts. This aspect is represented in Fig. 13. Using the MARTE *allocation*

Fig. 12 Refined functional level specification of the CCAS

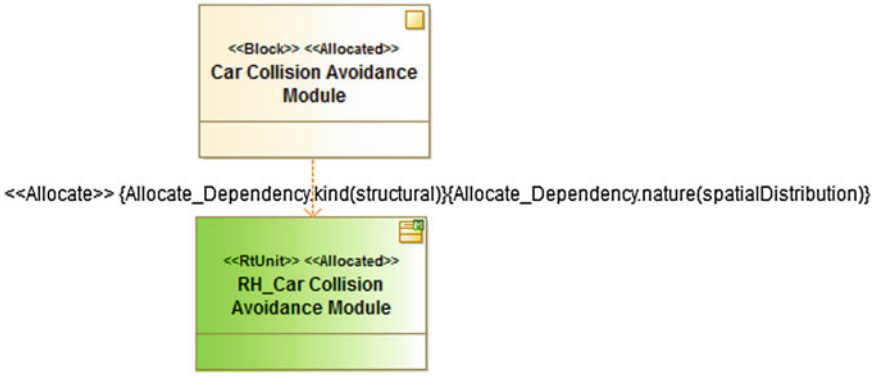


Fig. 13 Allocation between functional/refined functional level specifications

mechanism, we express that the allocation is *structural* (The structural aspects are thus related from source to target) and *spatial* in nature.

5.7 Clock Specification

Once the initial specification has been carried out, modeling of hardware and software aspects of the required functionality is possible in a parallel manner. For that purpose, we first create a "clock catalogue" (itself stored in the MADES CRP) using MARTE time concepts (which can be used to describe different timing aspects such as physical/logical or discrete/dense clocks etc.), as illustrated in Fig. 14, depicting the available clock elements (such as clock types) that are to be used by the execution platform of the CCAS. Here, an initial ideal clock type serves as the basis for the `Main` and `System` clock types. In this case study, all the clocks types are discrete in nature using the MARTE *Time* package, and their clock frequencies can be specified using the related tagged values (not visible in the figure).

Thus, three clock types are specified: an `IdealClock` (with a clock frequency of 50 MHz) that serves as base for the two other clock types, `SystemClock` and `HardwareClock` (with respective frequencies of 100 and 150 MHz). All modeled

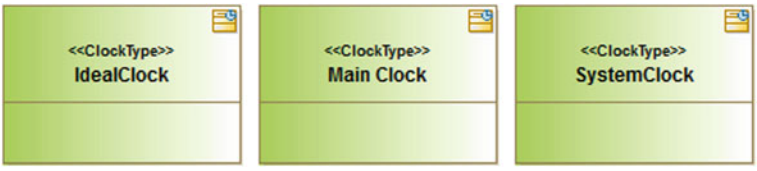


Fig. 14 Specification of clock types related to CCAS

concepts are appropriately stereotyped as `ClockType`. The `IdealClock` is the basic clock type present in the system running at a certain frequency, while the other two reference this basic clock and run at much higher frequencies.

5.8 Hardware Specification

At the hardware specification level, the abstract hardware concepts of the execution platform are modeled first as shown in Fig. 15. The abstract hardware modeling contains the controller for radar module along with its local memory; the image processing module and a shared memory, a system clock and other additional hardware resources (radar and camera modules, braking system, etc.); all of which communicate via a CAN bus.

The MARTE *GRM* package stereotypes are applied onto the different hardware resources: for example `ComputingResource` for the controller and the image processing module, `StorageResource` for the local and shared memories, `CommunicationMedia` for the `CAN` bus, while `DeviceResource` is used for the other hardware components. Here, the hardware specification also contains a clock `sysclk` of the type `SystemClock` specified earlier in Fig. 14. Here using the MARTE *Time* package, we add a clock constraint onto the clock, specifying that this clock (and related clock type) runs at a rate 10 times faster than that of the ideal clock (and the ideal clock type).

The hardware specification contains different hardware components which themselves are either further composed of sub components, or have internal behaviors,

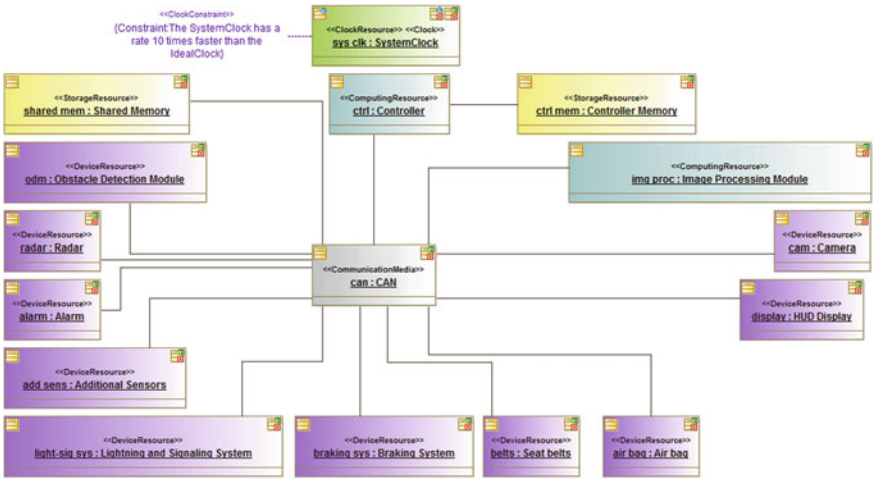


Fig. 15 Abstract hardware specification of CCAS

expressed by means of classic UML behavioral diagrams. We now describe the internal behavior of three hardware components:

In Fig. 16, we describe the internal behavior of the `Radar` component by means of a state machine diagram. The `RadarBehavior` state machine is stereotyped as `TimedProcessing` (not shown in the Figure). This permits to bind the processing of this behavior to time by means of a clock. Here the Radar remains in a single `receivingData` state and continues to send data to the controller at each tick of the `SystemClock`, every 100 ms.

In Fig. 17, the internal behaviour of the controller is specified. The controller contains three states, `noAction`, `warning` and `criticalwarning`. The controller initially remains in the `noAction` state when distance from incoming objects is greater than 3 m.

However, if the distance decreases to less than 3 m, then the controller switches to the `warning` state. If it remains in that particular state for 300 ms and distance is still less than 3 m but greater than 2 m, then a break interrupt is carried out and controller sends the normal brake command to the `Braking System`. Similarly, if distance decreases to less than 2 m, then the controller enters into a `criticalwarning` state. If it stays in that state for 300 ms and distance is still less than 2 m, then controller sends an emergency brake command to the `Braking System`.

Figure 18 displays the behavior of the `Braking System` when it receives commands from the controller. It normally remains in an `idle state` and depending upon a particular command received, switches to either the `normalBraking` or the `emergencyBraking` state. In a normal condition, the `Braking System`

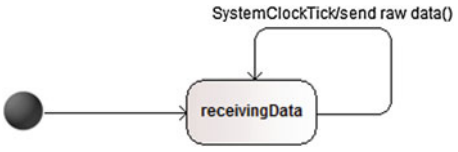


Fig. 16 Behavior of the radar module present in the CCAS

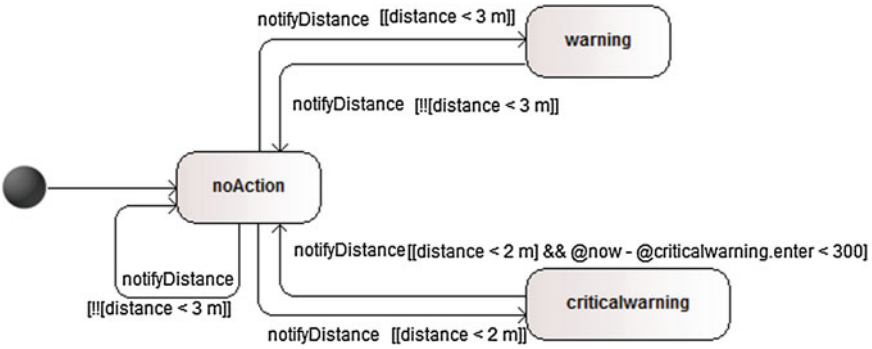


Fig. 17 Internal behavior of the CCAS controller

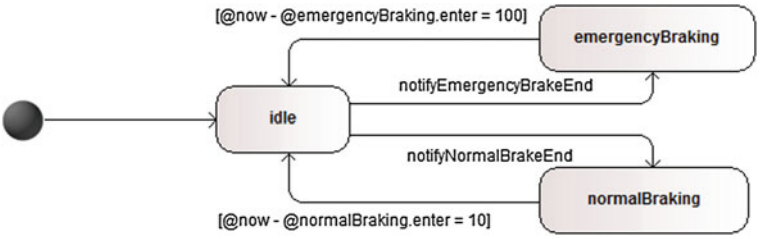


Fig. 18 Internal behavior of the braking system

applies brakes for 10 ms, while for an emergency condition, emergency brakes are applied for 100 ms.

5.9 Software Specification

We now turn towards modeling of the software specification of the execution platform of the CCAS, as displayed in Fig. 19. Here, schedulable tasks related to the hardware modules are modeled along with their communications. A scheduler is also present that manages the overall scheduling based on a fixed priority algorithm.

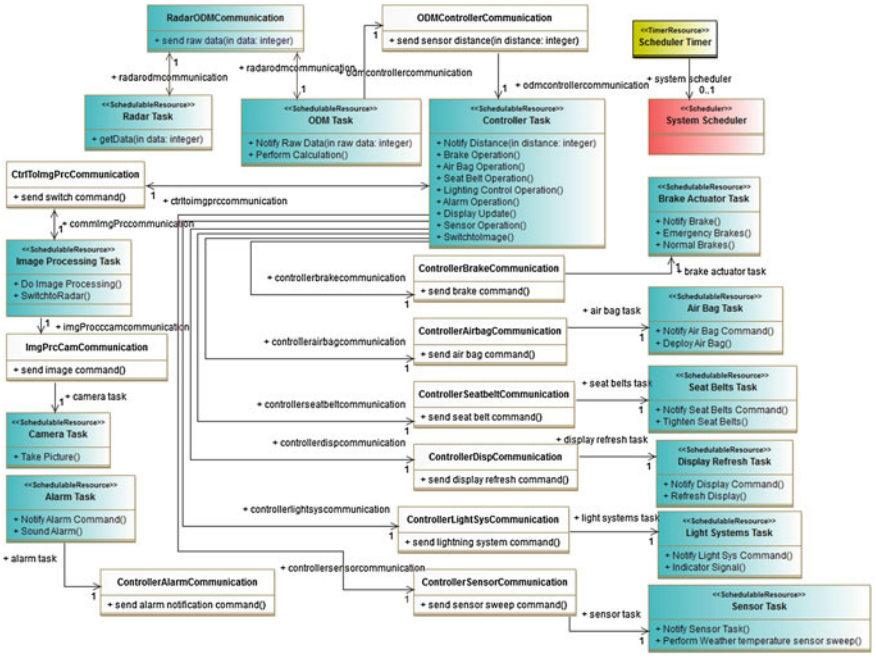


Fig. 19 Software specification of the CCAS (classes and their operations)

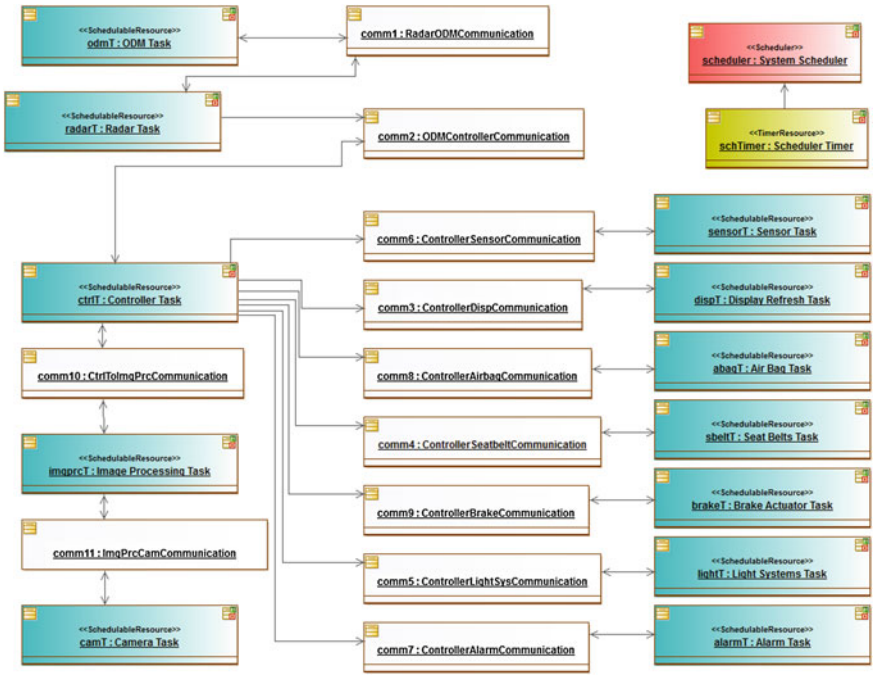


Fig. 20 Software specification of the CCAS (instance level)

The different tasks are stereotyped as `SchedulableResource`, indicating that they are scheduled by means of a `System Scheduler`, itself appropriately stereotyped as a `Scheduler`. Each task contains a number of operations, indicating the functionality related to that particular task. The software specification is also modeled at the instance level as illustrated in Fig. 20, for an eventual allocation between the software/hardware specifications.

5.10 Software to Hardware Allocation

Once the hardware and software specifications have been carried out, we carry out Co-Design allocations between the two using the MADES *Allocation Diagram*. This is done to map the CCAS software concepts to the hardware ones. Here in Fig. 21, the majority of the tasks (such as `Brake Actuator Task`, `Air Bag Task`) are allocated to the controller by means of a *temporal* allocation, while the `Radar` and `ODM` tasks are allocated to their respective hardware modules by means of *spatial* allocations (these properties are not shown in the Figure). While tasks related to the image processing module such as `Camera Task` are mapped on to it by means of a *temporal* allocation. Finally, all the communications are allocated to the `CAN` bus.

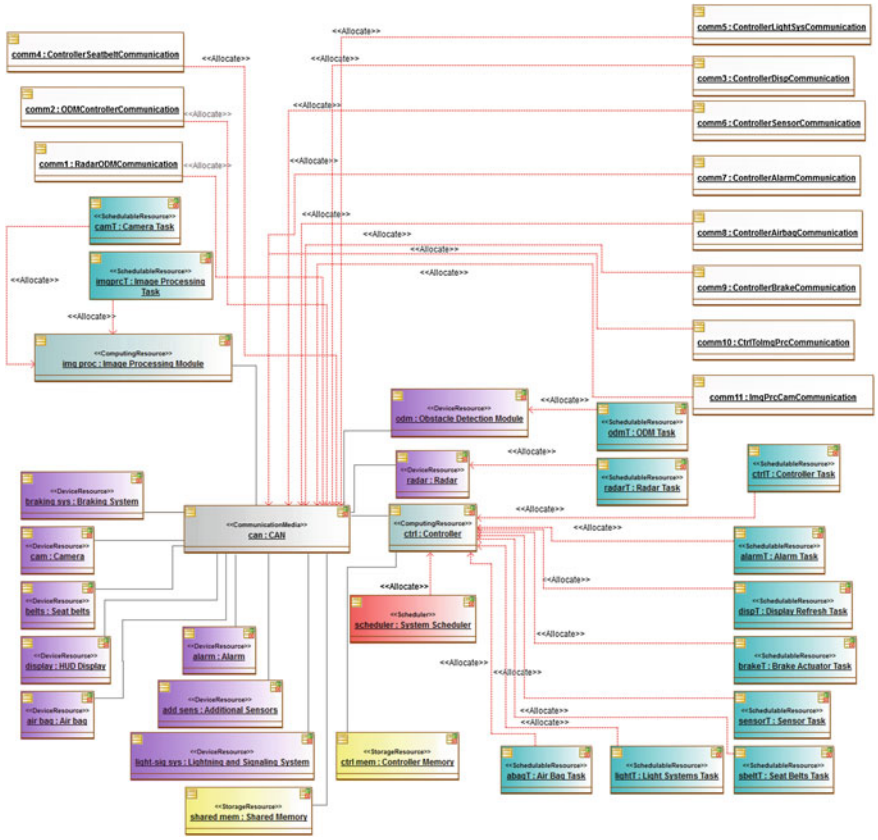


Fig. 21 Mapping software resources to the hardware modules of CCAS

It should be noted that while the *Allocated* stereotype on the software and hardware concepts has been applied similarly to the concepts illustrated in Fig. 13, they have not been displayed here for a better visualization.

We now move on to the detailed hardware and software specification design phases of the CCAS. Here, in the context of this book chapter, we only focus on a particular aspect of the CCAS, the `Obstacle Detection Module` and its corresponding task, which were initially specified in the abstract hardware/software design phases. We first describe the related enriched detailed hardware/software specifications, and then carry out the final mapping. In [29], another module of the CCAS, the `Image Processing Module` has been depicted with related detailed hardware/software specifications along with their allocation. Additionally, verification, code generation and synthesis on a Virtex V series FPGA has also been carried out regarding this module. These aspects are also included in another chapter of this book, dealing with MADES code generation aspects.

5.11 Detailed Hardware Specification

Once the initial abstract hardware specification has been modeled, the designer can move on to modeling of the detailed hardware specification which corresponds more closely to the actual implementation details of the execution platform. These detailed specifications may correspond to a simple one-to-one mapping to the abstract hardware specifications such as a `ComputingResource` being mapped to a `HwProcessor` for example, albeit with some additional details such as operating frequencies of processors, memory and address sizes for hardware memories, etc. It is also possible to enrich the detailed specifications with additional details (such as additional of behavior, internal structure etc.), as illustrated in Fig. 22 showcasing the enriched `HW Obstacle Detection Module`.

Here in the figure, the `HW Obstacle Detection Module` is itself stereotyped as a MARTE `HwComputingResource` and `mades_architecture`. All the components are automatically typed with MARTE and MADES stereotypes automatically, thanks to the mapping between the stereotype sets in Modelio. For example, the local `bram` memory is typed as a MARTE `HwMemory` and corresponding `mades_memory`, while the `gps` is dually stereotyped as `HwDevice` and `mades_ipcore`. The stereotypes having a prefix 'mades' denote concepts needed by the underlying model transformations, such as an iptype attribute of the mades_processingnode stereotype which tells the hardware generation transformation which IP core (and the version) to use for the modeled processor such as Microblaze or PowerPC processors. While it was possible to just add these concepts to MARTE stereotypes as an extension of the profile, the advantage offered by our approach is the MARTE profile remains intact and any underlying changes in the model transformations can be mapped to MARTE concepts, transparent to the end

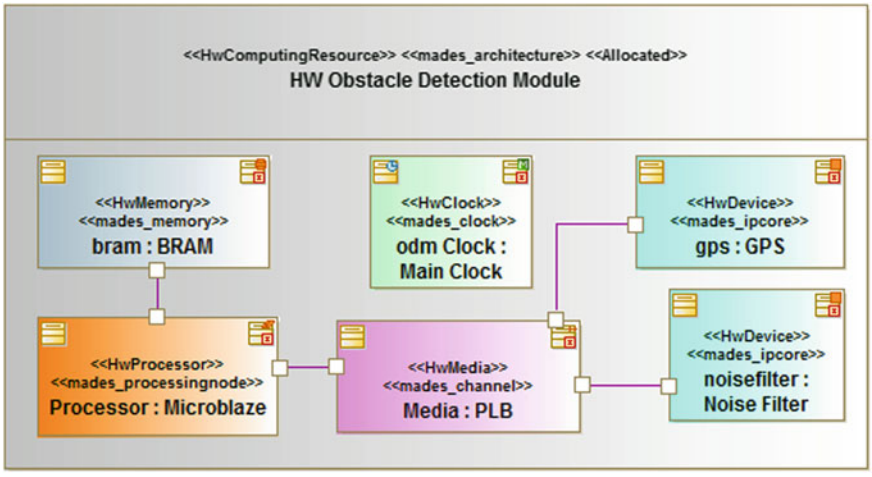


Fig. 22 Detailed hardware specification of the obstacle detection subsystem

user. For example, the MARTE concepts do not include the possibility of defining the type of a processor, such as a softcore or hardcore processor. These aspects can be added to the mades_processingnode stereotype and the model transformations accordingly, without changing the original MARTE specifications. Hence this approach also enables portability, as designers from other RTES industry and academia familiar with MARTE will be able to comprehend the specifications without needing to interpret another domain-specific language (DSL).[5]

5.12 Detailed Software Specification

In parallel, a designer can model the detailed software specification as seen in Fig. 23, which basically correspond to an enriched version of the ODM Task defined in Sect. 5.9. Here, the refined `SW Obstacle Detection Task` contains several threads along with their operations. The `nfilter` thread removes any noise from the incoming signal, while the `gps` thread calculates the position and velocity of the car containing the CCAS. The results are then sent to a `corr` thread that carries out a correlation and detects if there are any obstacles in the trajectory of the car with respect to its relative position. This data is then sent to an `output` thread which in turn sends this data to the controller of the CCAS.

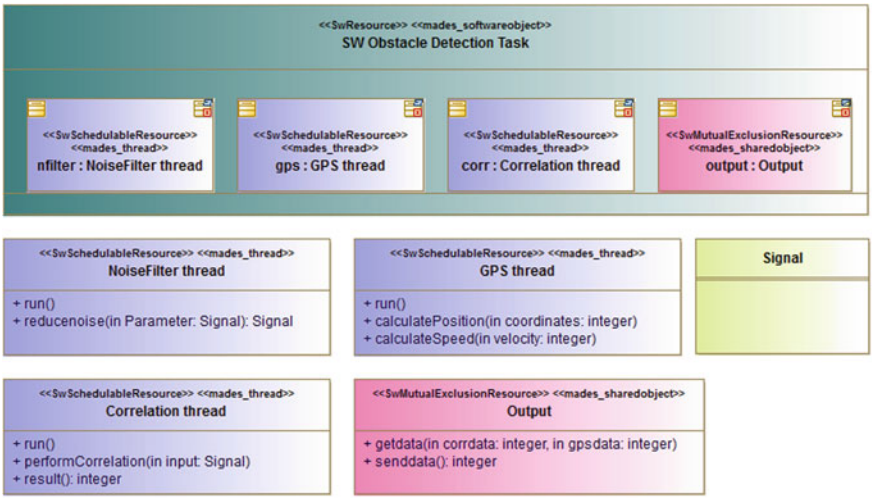


Fig. 23 Detailed software specification of the obstacle detection task

[5] Domain-Specific Language: http://en.wikipedia.org/wiki/Domain-specific_language.

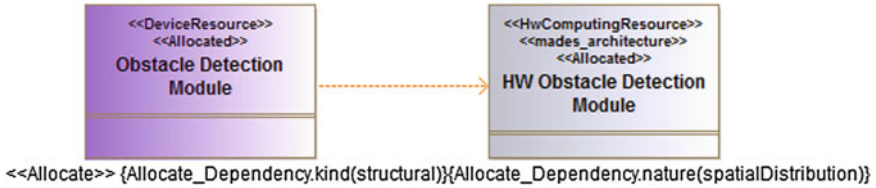


Fig. 24 Mapping of ODM related hardware/detailed hardware specifications

5.13 Allocating Hardware to Detailed Hardware Specifications

Once the detailed hardware specifications have been modeled, it is possible to carry out a refinement allocation that links the hardware to the detailed hardware specifications. In particular, it enables to move from abstract hardware specifications to detailed ones corresponding closely to an RTL (Register Transfer Level) implementation. In the specific case of CCAS and the obstacle detection aspects, we carry out a refinement allocation from the `Obstacle Detection Module` (and related instances) to the `HW Obstacle Detection Module` (and its instances), as shown in Fig. 24.

5.14 Allocation Software to Detailed Software Specifications

In a similar manner, the software specifications are refined and mapped onto the detailed software specifications, as shown in Fig. 25. Here, the `ODM Task` is refined to its detailed version, the `SW Obstacle Detection Module` via a refinement allocation.

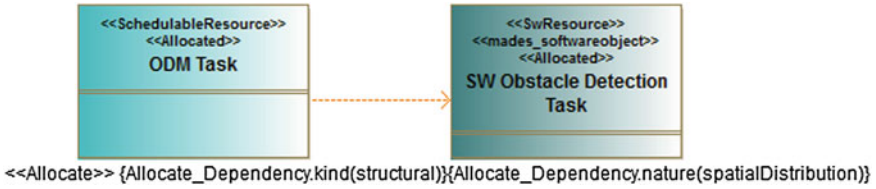


Fig. 25 Allocating software and detailed software specifications

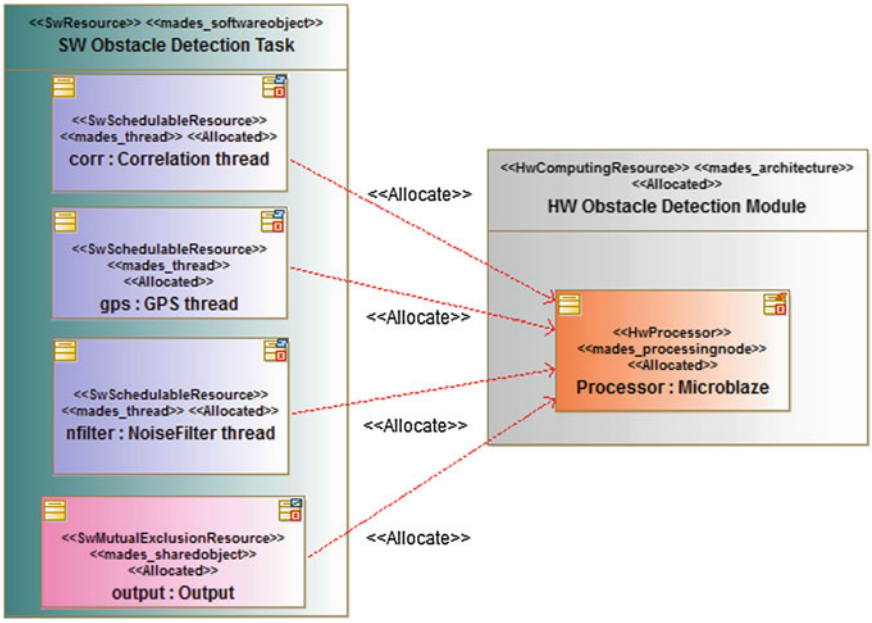


Fig. 26 Allocating the detailed software/hardware specifications of the ODM

5.15 Allocating Detailed Software to Detailed Hardware Specifications

Finally, once all the detailed specifications related to the software and hardware aspects of the obstacle detection subsystem have been modeled, it is possible to carry out a final allocation from the detailed software to the detailed hardware specifications. Here, as seen in Fig. 26, the different threads are spatially allocated onto the single Processor present inside the Hw Obstacle Detection Module.

Once this final design phase is completed, it is possible to carry out the subsequent phases of the MADES methodology, such as code generation and implementation in execution platforms. However, these steps have not been mentioned in this particular context, and are the scope of the chapter dealing with MADES model transformations and code generation.

6 Lessons Learned from the CCAS Project and Future Research Directions

Following the proposed MADES methodology, we demonstrated how system designers were able to use MADES's SysML/MARTE subset within a given work flow and with the aid of a modeling tool such as Modelio, in the particular case of

CCAS. The MADES methodology design phases can help and guide designers to follow a flexible and generic work flow for their eventual case studies, and provided semantics to the usage of UML, SysML and MARTE standards.

Particularly, inclusion of unique MADES diagrams for each MADES design phase comprising of either SysML or MARTE concepts (depending on the design phase) and a set of unique command set decreased the overall the design time and the learning curve, as compared to usage in expert mode. Here expert mode refers to annotating UML concepts (such as classes, instances, ports etc.) with the target profile concepts, as found in traditional modeling practices present in normal open source or commercial UML CASE tools [37, 38]. Using Modelio or other modeling tool in expert mode for complex profiles like SysML and MARTE was found to be a very cumbersome task, as the user has to first create UML concepts, such as classes, instances and ports and then annotate them using the profiles accordingly. Additionally, utilization of the profiles directly involved a lot of guess work in the cases where the designer was not familiar with the profiles, resulting in arisal of significant design errors, due to annotation of UML modeling concepts with incompatible profile concepts.

Therefore, usage of MARTE and SysML via MADES diagrams was found to be much more easier and intuitive. In cases when the same MARTE concept could be applied to different UML elements (classes, instances, connectors, ports, etc.) the diagrams were able to guide the system designers. A concrete example can be given on the `HwProcessor` stereotype, present in the MARTE profile that corresponds to a processor at the detailed hardware modeling design phase. In MARTE and normal UML CASE tools and editors, this stereotype can be rightly annotated to different UML modeling elements, such as classes, instances but incorrectly to ports as well, which does not makes sense from a hardware designer's point of view. In MADES the designers are guided by avoiding this mistake by only mapping the HwProcessor stereotype to UML classes and instances, and this command is available in the detailed hardware specification diagram as seen in Fig. 3 and appropriately named as a 'Processor'. In this way, the designers do not have to be concerned with the MARTE or UML concepts, and they can just select the hardware concepts available as commands present in Modelio, and then carry out modeling according to their design specifications.

The MADES CRP satisfied the needs of the MADES end users and task evaluators. The evaluators were able to store manually as much information as desired for keeping track of the various developed versions of the components as IPs in the CRP. The evaluators stored for each component: its name, description, version, keywords and tag, developer name, tool used for development and so on.

The integration of the CRP with a modeling environment such as Modelio is also a significant contribution. Currently, final integration is in process between Modelio and the CRP, which will enable designers to develop their component based IPs, which can be in turn automatically stored in the CRP; enabling IP-reuse when designers need to create systems requiring these components.

7 Conclusions

This chapter aims to present a complete methodology integrated in the MADES EU FP7 project project, for the design and development of real-time embedded systems using an effective subset of UML profiles: SysML and MARTE. we present our contributions by proposing an effective subset of the two profiles, forming the basis of MADES language and propose related set of unique diagrams. While both profiles provide numerous concepts and supporting tools, in the scope of the MADES project, the specific set of diagrams help to increase design productivity, decrease production cycles and promote synergy between the different designers/teams working at different domain aspects of the global system in consideration. These diagrams have been developed after careful analysis and provide only those effective SysML or MARTE stereotypes that are needed for a particular design phase. However, the designer also has a choice to select all available stereotypes present in the two profiles, thus enabling him to either work in an expert mode, or be guided via the MADES language subset. Thus, our MADES methodology could inspire future revisions of the SysML and MARTE profiles and may eventually aid in their evolution. This methodology is generic in nature and can be applied to other studies and projects focusing on high abstraction based design specifications. Finally, the different language concepts and associated diagrams in the methodology have been illustrated in a case study related to a car collision avoidance system.

Acknowledgments This research presented in this paper is funded by the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement No. 248864 (MADES). The authors would like to thank all of the MADES partners for their valuable inputs and comments.

References

1. OMG: Portal of the Model Driven Engineering Community (2007), http://www.planetmde.org
2. Object Management Group Inc.: Omg unified modeling language (OMG UML), superstructure, v2.4.1 (2011), http://www.omg.org/spec/UML/2.4.1
3. S. Sendall, W. Kozaczynski, Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
4. A. Bagnato et al., MADES: Embedded systems engineering approach in the avionics domain. First workshop on hands-on platforms and tools for model-based engineering of embedded systems (HoPES) (2010)
5. MADES: EU FP7 Project (2011), http://www.mades-project.org/
6. Object Management Group Inc.: Final Adopted OMG SysML Specification (2012), http://www.omg.org/spec/SysML/1.3/
7. OMG: Modeling and Analysis of Real-time and Embedded systems (MARTE) (2011), http://www.omg.org/spec/MARTE/1.1/PDF
8. OMG: Object Management Group (2012), http://www.omg.org/
9. M. Faugere, T. Madeleine, R. Simone, S. Gerard, in *MARTE: Also an UML Profile for Modeling AADL Applications*. ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, IEEE Computer Society (2007), pp. 359–364
10. H. Espinoza, An Integrated Model-Driven Framework for Specifying and Analyzing Non-Functional Properties of Real-Time Systems, PhD thesis, University of Evry, FRANCE, 2007

11. C. André, A. Mehmood, F. Mallet, R. Simone, Modeling SPIRIT IP-XACT in UML-MARTE. MARTE workshop on design automation and test in Europe (DATE) (2008)
12. A. Koudri et al., Using MARTE in the MOPCOM SoC/SoPC Co-Methodology. MARTE workshop at DATE'08 (2008)
13. EDIANA: ARTEMIS project (2011), http://www.artemis-ediana.eu/
14. TOPCASED: The Open Source Toolkit for Critical Systems (2010), http://www.topcased.org/
15. W. Mueller et al., The SATURN Approach to SysML-based HW/SW Codesign, in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (2010)
16. D.D. Gajski, R. Khun, New VLSI tools. IEEE Comput. **16**, 11–14 (1983)
17. M. Mura et al., Model-based design space exploration for RTES with SysML and MARTE, in *Forum on specification, verification and design languages (FDL 2008)* (2008), pp. 203–208
18. Information Society Technologies, OMEGA: Correct Development of Real-Time Embedded Systems (2009), http://www-omega.imag.fr/
19. L. Ober et al., Projet Omega: Un profil UML et un outil pour la modelisation et la validation de systemes temps reel. **73**, 33–38 (2005)
20. OMG: UML Profile For Schedulability, Performance, and Time (2012), http://www.omg.org/spec/SPTP/
21. INTERESTED: EU FP7 Project (2011), http://www.interested-ip.eu/index.html
22. H. Espinoza et al., Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems, in *ECMDA-FA'09* (Springer, 2009), pp. 98–113
23. D.S. Kolovos et al., Eclipse development tools for Epsilon. Eclipse Modeling Symposium on Eclipse Summit Europe (2006)
24. N. Matragkas et al., D4.1: Model transformation and code generation tools specification. Technical report (2010), http://www.mades-project.org/
25. L. Baresi et al., D3.1: Domain-specific and User-centred Verification. Technical report (2010), http://www.mades-project.org/
26. L. Baresi et al., D3.3: Formal Dynamic Semantics of the Modelling Notation. Technical report (2010), http://www.mades-project.org/
27. Zot: The Zot bounded model/satisfiability checker (2012), http://zot.googlecode.com
28. I. Gray, N. Audsley, Exposing non-standard architectures to embedded software using compile-time virtualisation. International conference on compilers, architecture, and synthesis for embedded systems (CASES'09) (2009)
29. A. Bagnato et al., D1.7: MADES Final Approach Guide. Technical report (2012), http://www.mades-project.org/
30. I. Gray et al., Model-based Hardware Generation and Programming—The MADES Approach. 14th International Symposium on Object and Component-Oriented Real-Time Distributed Computing Workshops (2011)
31. Modelio: Open source UML Editor and MDE Workbench (2012), www.modelio.org
32. Xilinx: MicroBlaze Soft Processor Core (2011), http://www.xilinx.com/tools/microblaze.htm
33. I.R. Quadri et al., D1.6: MADES Tool Set—Final Version. Technical report (2012), http://www.mades-project.org/
34. Softeam: Modeliosoft: Modelio Community Portal (2012), http://www.modeliosoft.com/en.html
35. L. Baresi et al., Newblock D3.2: Models and Methods for Systems Environment. Technical report (2012), http://www.mades-project.org/
36. OpenModelica: Open-source Modelica-based modeling and simulation environment (2012), http://www.openmodelica.org/
37. NoMagic: Magic Draw: Architecture made simple (2012), http://www.magicdraw.com/
38. Papyrus: Eclipse Project on an Open source UML editor (2012), http://www.eclipse.org/modeling/mdt/papyrus/

Test-Driven Development as a Reliable Embedded Software Engineering Practice

Piet Cordemans, Sille Van Landschoot, Jeroen Boydens and Eric Steegmans

Abstract Due to embedded co-design considerations, testing embedded software is typically deferred after the integration phase. Contrasting with the current embedded engineering practices, Test-Driven Development (TDD) promotes testing software during its development, even before the target hardware becomes available. Principally, TDD promotes a fast feedback cycle in which a test is written before the implementation. Moreover, each test is added to a test suite, which runs at every step in the TDD cycle. As a consequence, test-driven code is well tested and maintainable. Still, embedded software has some typical properties which impose challenges to apply the TDD cycle. Essentially, uploading software to target is generally too time-consuming to frequently run tests on target. Secondary issues are hardware dependencies and limited resources, such as memory footprint or processing power. In order to deal with these limitations, four methods have been identified and evaluated. Furthermore, a number of relevant design patterns are discussed to apply TDD in an embedded environment.

P. Cordemans (✉) · S. Van Landschoot · J. Boydens
KHBO Department of Industrial Engineering Science and Technology, Zeedijk 101, B-8400 Ostend, Belgium
e-mail: Piet.Cordemans@kuleuven.be

S. Van Landschoot
e-mail: Sille.VanLandschoot@vives.be

J. Boydens
e-mail: Jeroen.Boydens@kuleuven.be

E. Steegmans
KU Leuven Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium
e-mail: Eric.Steegmans@cs.kuleuven.be

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_4,

1 Test-Driven Development

Test-Driven Development (TDD) is a fast paced incremental software development strategy, based on automated unit tests. First, this section describes the rationale of TDD (Sect. 1.1). Then, it describes the core of TDD (Sect. 1.2). Next, the advantages and difficulties (Sect. 1.3) of developing by TDD are discussed. Finally, an overview of unit testing frameworks (Sect. 2) is given.

1.1 TDD Rationale

As embedded systems are currently becoming more complex, the importance of their software component rises. Furthermore, due to the definite deployment of embedded software once it is released, it is unaffordable to deliver faulty software. Thorough testing is essential to minimize software bugs. The design of embedded software is strongly dependent on the underlying hardware. Co-design of hardware and software is essential in a successful embedded system design. However, during the design time, the hardware might not always be available, so software testing is often considered to be impossible. Therefore testing is mostly postponed until after hardware development and it is typically limited to debugging or ad-hoc testing. Moreover, as it is the last phase in the process, it might be shortened when the deadline is nearing. Integrating tests from the start of the development process is essential for a meticulous testing of the code. In fact, these tests can drive the development of software, hence Test-Driven Development.

It is crucial for embedded systems that they are tested very thoroughly, since the cost of repair grows exponentially once the system is taken in production, as stated in Fig. 1, which depicts the law of Boehm [1]. However, the embedded system can only be tested once the development process is finished. In a waterfall-like strategy for developing embedded systems, the testing phase is generally executed manually. This ad-hoc testing is mostly heuristic and only focuses on one specific scenario. In order to start testing embedded as early as possible, a number of problems arise. One problem is hardware being unavailable early on in the development proces, and another is the difficulty to automatically test embedded systems.

1.2 TDD Mantra

Test-Driven Development [2] consists of a number of steps, sometimes called the TDD mantra. In TDD, before a feature is implemented, a test is written to describe the intended behavior of the feature. Next, a minimal implementation is provided to get the test passing. Once the test passes, code can be refactored. Refactoring is restructuring code without altering its external behavior or adding new features

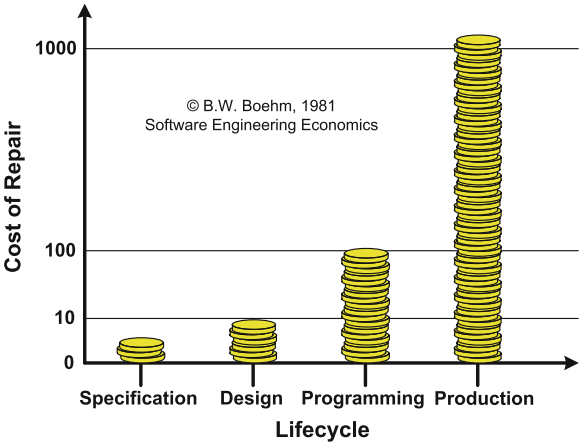
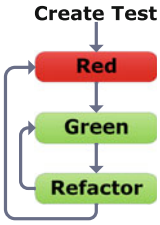


Fig. 1 Law of Boehm

Fig. 2 TDD mantra



to it [3]. When the quality of code meets an acceptable level, the cycle starts over again, as visually represented in Fig. 2.

TDD reverses the conventional consecutive order of steps, as tests should be written before the code itself is written. Starting with a failing test gives an indication that the scope of the test encompasses new and unimplemented behavior. Moreover, if no production code is written without an accompanying test, one can assure that most of the code will be covered by tests.

Also fundamental to the concept is that every step is supported by executing a suite of automated unit tests. These tests are executed to detect regression faults either due to adding functionality or refactoring code.

Fundamental to the concept of TDD is that refactoring and adding new behavior are strictly separated activities. When refactoring, tests should remain passing. Yet should a failure occur, it should be solved in quick order or the changes must be reverted. On the other hand, when adding new functionality, the focus should stay on the current issue, only conducting the refactorings when all tests are passing. Refactoring can and should be applied to the tests themselves as well. In that situation the implementation stays the same and can be reassured that the test does not change its own scope.

The properties of a good unit test can be described by the F.I.R.S.T. acronym, which is coined by Martin [4].

1. Fast: the execution time of the tests should be limited. If it takes too long, a suite of many tests will limit development speed.
2. Independent: the setup and result of a test should be independent of other tests. Dependencies between tests complicate execution order and lead to failing tests when changing or removing tests.
3. Repeatable: the execution of a test should be repeatable and deterministic. False positives and negatives lead to wrong assumptions.
4. Self-validating: the result of a test should lead to a failing or passing assertion. This may sound obvious, nevertheless should the test lead to something else, like a log file, it cannot be verified automatically.
5. Timely: this refers to the TDD way of writing tests as soon as possible.

1.3 Advantages and Difficulties

Programming according to the principles of TDD has a number of advantages. First and foremost are those which result from incrementally developing an automated test suite. Also TDD allows for a steady and measurable progression. Finally TDD forces a programmer to focus on three important aspects.

Frequent testing

As TDD imposes to frequently run a test suite, four particular advantages result from it. First, the tests provide a safety net when refactoring, alerting the programmer when a refactoring went wrong, effectively altering the behavior of software. Next, running tests frequently will detect regression when code for a new feature interferes with other functionality. Furthermore, when encountering a bug later on (TDD cannot guarantee the absences of bugs in code), a test can be written to detect the bug. This test should fail first, so one can be sure it tests the code where the bug resides. After that making the test pass will solve the bug and leave a test in place in order to detect regression. Moreover, tests will ensure software modules can run in isolation, which improve their reusability. Finally, a test suite will indicate the state of the code and when all tests are passing, programmers can be more confident in their code.

Steady development

Next to the automated test suite, TDD also allows for a development rate, which is steady and measurable. Each feature can be covered by one or more tests. When the tests are passing, it indicates that the feature has been successfully implemented. Moreover, through strategies like faking it, it becomes possible to adjust the development rate. It can go fast when the implementation is obvious or slower when it becomes difficult. Anyhow, progression is assured.

Encapsulation

Finally TDD is attributed to put the focus on three fundamental issues. First, focus is placed on the current issue, which ensures that a programmer can concentrate on one thing at a time. Next, TDD puts the focus on the interface and external behavior of software, rather than its implementation. By testing its own software, TDD forces a programmer to think how software functionality will be offered to the external world. In this respect, TDD is complementary to Design by Contract where a software module is approached by a test case instead of formal assertions. Lastly TDD moves the focus from debugging code to testing. When an unexpected issue arises, a programmer might revert to an old state, write a new test concerning an assumption and see if it holds. This is a more effective way of working as opposed to relying on a debugger.

TDD has a number of imperfections, which mainly concern the overhead introduced by testing, thoroughness of testing and particular difficulties when automating particular hard to test code.

Overhead

Writing tests covering all development code doubles the amount of code that needs to be written. Moreover, TDD is specifically effective to test library code. This is code which is not directly involved with the outside world, for instance the user interface, databases or hardware. However when developing code related to the outside world, one has to lapse on software mocks. This introduces an additional overhead, as well as assumptions on how the outside world will react. Therefore it becomes vital to do some manual tests, which verify these assumptions.

Test coverage

Unit tests will only cover as much as the programmer deemed necessary. Corner cases tend to be untested, as they will mostly cover redundant paths through the code. In fact writing tests which will cover the same path with different values are prohibited by the rule that a test should fail first. This rule is stated with good reason, as redundant tests tend to lead to multiple failing tests if regression is introduced, hence obfuscating the bug. Testing corner case values should be done separately from the activity of programming according to TDD. An extra test suite, which is not part of the development cycle allows for a minimalistic effort to deal with corner case values. Should one of these tests detect a bug, the test can easily be migrated to the TDD test suite to fix the problem and detect regression.

On the other hand programmers become responsible to adhere strictly to the rules of TDD and only implement a minimum of code necessary to get a passing test. Especially in conditional code, one could easily introduce extra untested cases. For instance an if clause should only lead to an else clause, if a test demands to do so.

Furthermore, a consecutive number of conditional clauses tend to increase the number of execution paths without demanding to write extra test cases. Similar to the corner case values, an extra test suite can deal with this problem. However, TDD also encourages avoiding this kind of code, by demanding isolation. This will typically lead to a large number of small units, for instance classes, rather than one big complicated unit.

Next a critical remark has to be made on the effectiveness of the tests written in TDD. First, they are written by the same person who writes the code under test. This situation can lead to narrow focused tests, which only expose problems known to the programmer. In effect, having a large test suite of unit tests, does not take away the need of integration and system tests. On the other hand code coverage is not guaranteed. It is the responsibility of the programmer to diverge from the happy path and also test corner cases. Additionally, tests for TDD specifically focus on black box unit testing, because these tests tend to be less brittle than tests, which also test the internals of a module. However for functional code coverage glass box tests are also necessary.

Finally, a unit test should never replicate the code that it is testing. Replication of code in a test leads to a worthless test as bugs introduced in the actual code will be duplicated in the test code. In fact, code complexity of tests should always be less than the code under test. Moreover the tests that are written need to be maintained as well as production code. Furthermore setting up a test environment, might require additional effort, especially when multiple platforms are targeted.

The evaluation of the TDD strategy has been subject of multiple research projects. George and Williams [5] have conducted a research on the effects of TDD on development time and test coverage. Siniaalto [6] provides an overview of the experiments regarding TDD and productivity. Nagappan [7] describes the effects of TDD in four industrial case studies. Muller and Padberg [8] claim that the lifecycle benefit introduced by TDD outweighs its required investment. Note that research on the alleged benefits of TDD for embedded software is limited to several experience reports, such as written by Schooenderwoert [9, 10] and Greene [11].

2 Embedded Unit Testing Frameworks

In TDD the most valuable tool, is a unit testing framework [12]. Most of these frameworks are based upon an archetypal framework known as xUnit, from which various ports exist, like JUnit for Java and CppUnit for C++. In fact for C and C++ more than 40 ports exist to date and most of them are open source. Regardless of the specific implementation, most of these have some common structure, which is shown in Fig. 3.

A unit testing framework consists of a library and a test runner. On the one hand the library mostly provides some specific assertions, like checking equality for various types. Optionally it might also check for exceptions, timing, memory leaks, etc. On the other hand the test runner calls unit tests, setup, teardown and reports to the programmer. Setup and teardown in combination with a test is called a test fixture. First, setup provides the necessary environment for the unit test to execute. After test execution, teardown cleans the environment. Both are executed accordingly in order to guarantee test isolation. Instead of halting execution when it encounters a failing assertion, the test runner will gracefully eject a message, which contains valuable information of the failed assertion. That way all tests can run and a report is composed of failing and passing tests. For organizational reasons tests might be grouped into suites, which can be independently executed.

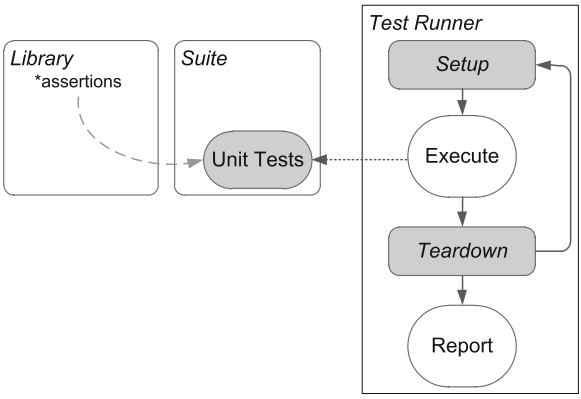


Fig. 3 xUnit testing framework

Choosing a unit test framework depends on three criteria [13].

- Portability. Considering the different environments for embedded system applications, portability and adaptability are a main requisite for a unit test framework. It should have a small memory footprint, allow to easily adapt its output and do not rely on many libraries.
- Overhead of writing a new test. Writing tests in TDD is a common occurrence and therefore it should be made easy to do so. However some frameworks demand a lot of boilerplate code, especially when it is obligatory to register each test. As registration of tests is quite often forgotten, this can lead to confusion.
- Minor features, such as timing assert functionality, memory leak detection and handling of crashes introduced by the code under test, are not essential, but can provide a nice addition to the framework. Furthermore, some unit testing frameworks can be extended with a complementary mocking framework, which facilitates the creation of mocks.

Deciding on a unit test framework is a matter of the specifications of the target platform. A concise overview is given.

- MinUnit [14] is the smallest C framework possible. It consists of two C macro's, which provide a minimal runner implementation and one assertion. This framework can be ported anywhere, but it should be extended to be of any use and it requires a lot of boilerplate code to implement the tests.
- Embunit [15] is a C based, self-contained framework, which can be easily adapted or extended. However, it requires to register tests, which is error-prone and labor intensive.
- Unity [16] is similar to Embunit and additionally contains a lot of embedded specific assertions. It is accompanied with a mocking framework and code generation tools written in Ruby to deal with boilerplate code.
- UnitTest++ [17] is C++ based, which can be ported to any but the smallest embedded platforms. Usability of the framework is at prime, but it requires some work to adapt the framework to specific needs.

- CppUTest [18] is one of the latest C++ testing frameworks, it also has a complementary mocking framework, called CppUMock.
- GoogleTest [19] is the most full-blown C++ unit test framework to date. It provides integration with its mocking framework, GoogleMock. However it is not specifically targeted to embedded systems and is not easily ported.

2.1 Mocking Hardware

Fundamental to automated testing of embedded software is to replace hardware dependencies with software mock [20] representations. Switching between the real and mock configurations should be effortless. Therefore hardware mocks must be swapped with the real implementation without breaking the system or performing elaborate actions to setup the switch. Five techniques have been identified, three based upon object oriented principles and three C-based, which facilitate the process.

2.1.1 Interface Based Mock Replacement

In the interface based design, as shown in Fig. 4, the effective hardware and mock are addressed through a unified abstract class, which forms the interface of the hardware driver. Calls are directed to the interface thus both mock and effective hardware driver provide an implementation. The interface should encompass all methods of the hardware driver to ensure compatibility. Optionally the mock could extend the interface for test modularity purposes. This enables customizing the mock on a test-per-test basis, reducing duplication in the test suite.

It should be noted that the interface could provide a partial implementation for the hardware independent methods. However, this would indicate that hardware dependencies are mixed with hardware independent logic. In this situation a refactoring is in order to isolate hardware dependent code.

Inheriting from the same interface guarantees compatibility between mock and real hardware driver, as any inconsistency will be detected at compile time. Regarding future changes, extending the real driver should appropriately reflect in the interface.

The main reason of concern with this approach is the introduction of late binding, which inevitably slows down the system in production. However it should be noted that such an indirection is acceptable in most cases.

2.1.2 Inheritance Based Mock Replacement

Figure 5 provides the general principal of inheritance based mock replacement. Basically, the real target driver is directly addressed. Yet as the mock driver inherits from the real target driver, it is possible to switch them according to their respective environment. However it requires that all hardware related methods are identified,

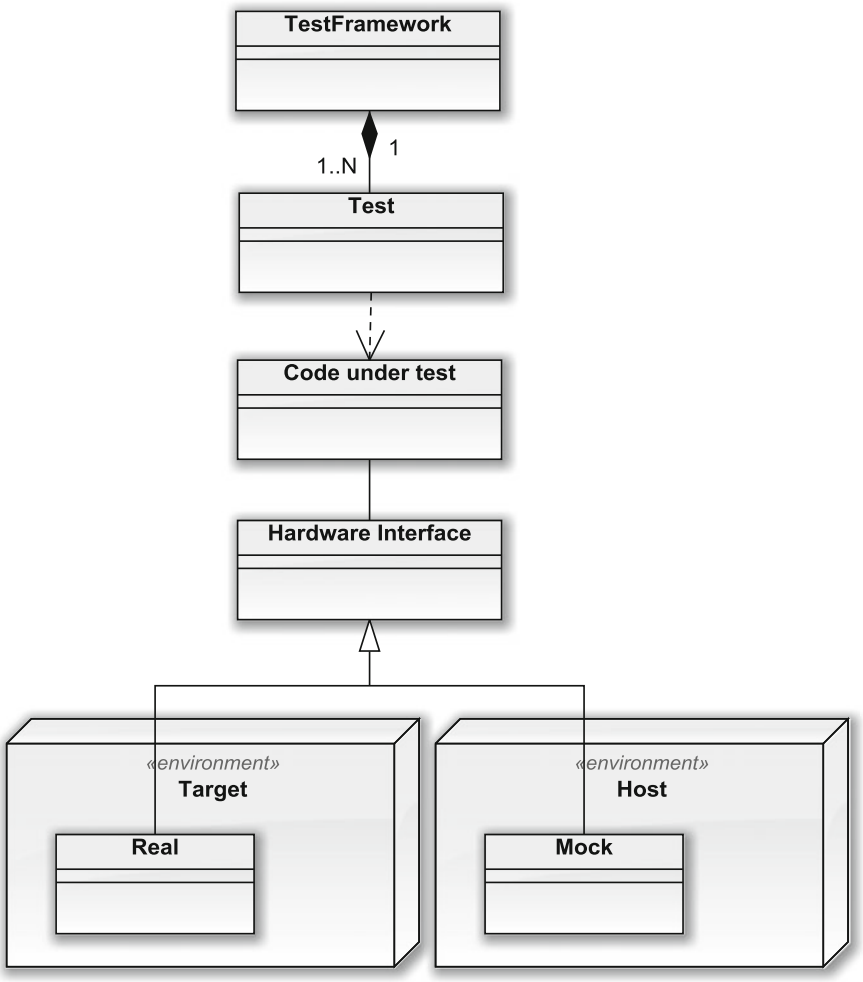


Fig. 4 UML class diagram of interface based mock replacement in different environments

at least given protected member access and are declared as virtual. These conditions allow overriding hardware related methods with a mock implementation on host.

However, these issues can be worked around with some macro preprocessing. First, all private members can be adjusted to public access solely for testing purposes. Also the virtual keyword can be removed in the target build.

Inheritance-based mock introduction is more about managing testability of code than actual testable design. That being said, all overhead of testability can be easily removed in production code. However, ensuring that the macro definitions do not wreck havoc outside the file is fundamental in this approach. Nonetheless, also when dealing with legacy code this approach is preferable. Considering the amount of

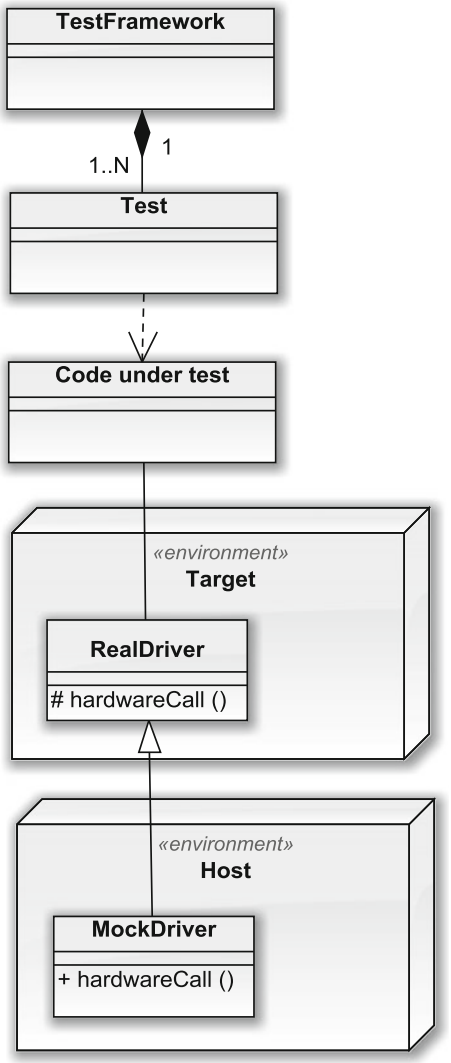


Fig. 5 UML class diagram of inheritance based mock replacement

refactoring, which is necessary to extract the hardware independent interface in the interface-based approach, the adjustments for inheritance-based mock replacement can be introduced without a layer of indirection.

2.1.3 Composition Based Mock Replacement

Interface or inheritance-based mock replacement realizes calling the effective methods in a uniform manner. Nevertheless, the correct mock should be installed in the first place, in order to be able to address hardware or mock functionality without overhead of managing dependencies. Therefore a composition based design pattern, called Dependency Injection, allows to manage these dependencies in a flexible manner.

Three different types of Dependency Injection [21] exist. First is constructor injection, in which a member object reference is injected during object construction of the composite object. The ConstructorInjection class

```
class ConstructorInjection {
public: ConstructorInjection(HardwareDependency* hw)
{ m_hw = hw };
virtual ~ConstructorInjection();
private:    HardwareDependency* m_hw;
};
```

shows a reference design of constructor injection. Next is setter injection,

```
class SetterInjection {
public:  SetterInjection();
virtual ~SetterInjection();
void injectDependency (HardwareDependency* hw)
{ m_hw = hw };
private:    HardwareDependency* m_hw;
};
```

which performs the same operation as constructor injection. Yet, instead of the constructor a setter method is used to register the reference. This introduces the possibility to change the reference at run-time without creating a new composite object. On the other hand, when compared to constructor injection, it requires an additional overhead in test management, namely the call of the setter method itself. Forgetting to do so will lead to incorrect initialized objects under test. Moreover setter injection will introduce additional overhead in the setup of the system in production. Considering real-time systems or multiple run-time creation and cleanup of the objects, the overhead becomes critical. Especially when considering resource constrained systems like embedded processors. Therefore a sound advice is to only use setter injection when its flexibility is required and otherwise use constructor injection by default.

Finally, interface injection registers the dependency by inheriting from an abstract class, which contains a setter method. Two approaches can be followed with interface injection. On the one hand, a specific class can be made for each type of objects to be injected. On the other hand, a generic interface class can be provided, which allows injecting objects of all types.

The previous strategies were based on object-oriented features, however embedded software is often written in C. The following techniques do not use OO features, yet allow switching unnoticeable, at least in production, between real and mock code.

2.1.4 Link-Time Based Mock Replacement

First is link-time based mock replacement [22], also known as link-time polymorphism or link-time substitution. The idea is to provide a single interface of functions in a header file and use the linking script or IDE to indicate which implementation file corresponds to it, i.e. the file containing the actual implementation or a similar file containing the mock implementation. Correspondingly the host build will refer to the mock files and the target build to the real implementation files, as visually represented in Fig. 6.

Practically the linker script (or IDE) will refer to three different subfolders. First is the common folder, which contains all platform independent logic as well as header files containing the hardware dependent function declarations. Next is the host folder, which will include the mocks and finally the target folder with the corresponding real implementations. Should a hardware implementation or mock file be missing,the linker will return an error message as a reminder. A practical example of the

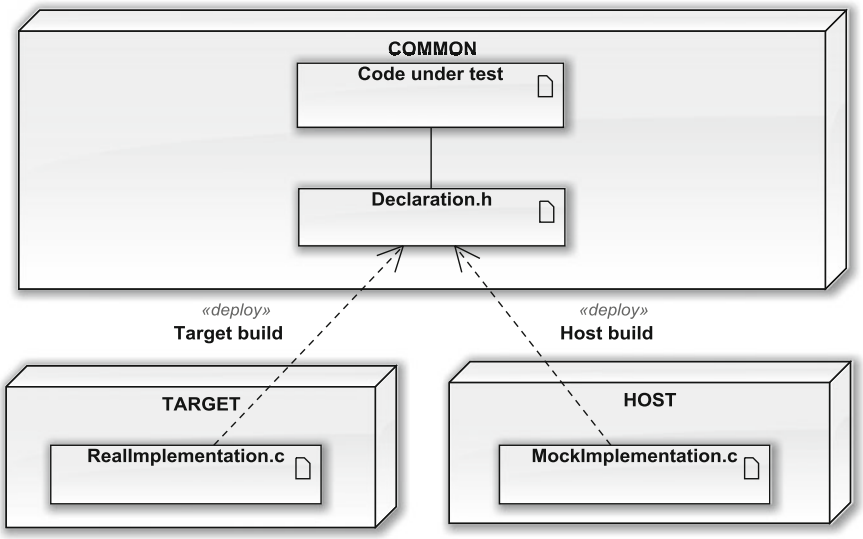


Fig. 6 Host and target build refer to the mock and real implementation file respectively

link-time based configuration is not given, considering the multitude of build systems and IDE's.

2.1.5 Macro Preprocessed Mock Replacement

While the link-time based macro replacement involves delivering the desired source code files to the linker, the macro preprocessed alternative involves preprocessor directives, which manipulate the source code itself. For instance,

```
#ifdef TESTONHOST
#include "mockdriver.h"
#else
#include "realdriver.h"
#endif
```

provides an almost identical effect to its link-time based alternative. Moreover, macro replacement allows to intersect inside a source file. First, the function call to be mocked is replaced by a new function definition. Also, the testing framework to implement the tests related to the code under test is injected in the file itself.

```
#ifdef TESTONHOST
#define functionMock(int arg1, int arg2)
            function(int arg1, int arg2)
void functionMock(int arg1, int arg2) {};
#endif

/* code containing function to be mocked */

#ifdef TESTONHOST
#include "unittestframework.h"
int main () {
/* run unit tests & report */
}
```

Although macros are commonly negatively regarded, the macros shown in the previous two listings are generally safe and will not lead to bugs which are hard to find. However the macro statements will pollute the source code, which leads to less readable and thus less maintainable code. The main advantage of macro preprocessed mock replacement is in dealing with legacy code. Capturing the behavior of legacy code in tests is something that should be done with the least refactoring, because in legacy code, tests are lacking to provide feedback on the safety of the refactoring operations. Using macros effectively allows leaving the production code unchanged, while setting up the necessary tests. Conversely, when developing new applications link-time based mock replacement is preferred, as it does not have any consequences on the production code.

2.1.6 Vtable Based Mock Replacement

Polymorphism can be obtained in C code by manually building the vtable [23]. However, implementing dynamic dispatch in C is not preferred when comparing it to either the preprocessing or link-time solution. Introducing the vtable in code results in an execution time overhead, which can be critical when considering the typical type of embedded C applications.

Furthermore constructing the vtable in C code is not preferred when C++ is a viable alternative. On the one hand, while C++ compilers can do extensive optimization on virtual functions where the actual type can be discovered at compile-time, this cannot be done by a C compiler. On the other hand, there is a manifold overhead in code management to implement the vtable system in C when compared to the native OO solution. In conclusion, the abstraction created by C++ allows to easily forget the overhead introduced by late binding, yet also permits to improve code maintainability.

3 Test-Driven Development for Embedded Software

Ideally Test-Driven Development is used to develop code which does not have any external dependencies. This kind of code suits TDD well, as it can be developed fast, in isolation and does not require a complicated setup. However, when dealing with embedded software the embedded environment complicates development. Four typical constraints influence embedded software development and have their effect on TDD. To deal with these issues four strategies have been defined [24–26], which tackle one or more of these constraints. Each of these strategies leads to a specific setup and influence the software development process. However, neither of these strategies is the ideal solution and typically a choice needs to be made depending on the platform and type of application.

3.1 Embedded Constraints

Development speed

TDD is a fast cycle, in which software is incrementally developed. This results in frequently compiling and running tests. However, when the target for test execution is not the same as the host for developing software, a delay is introduced into development. For instance, this is the time to flash the embedded memory and transmit test data back to the host machine. Considering that a cycle of TDD minimally consists of two test runs, this delay becomes a bottleneck in development according to TDD. A considerable delay will result in running the test suite less frequent, which in turn results to taking larger steps in development. This will introduce more

failures, leading to more delays, which in turn this will reduce the number of test runs, etc.

Memory footprint

Executing TDD on a target platform burdens the program memory of the embedded system. Tests and the testing framework are added to the program code residing in target memory. This results in at least doubling the memory footprint needed.

Cross-compilation issues

In respect of the development speed and memory footprint issues, developing and testing on a host system solves the previously described problems. However, the target platform will differ from the host system, either in processor architecture or build tool chain. These issues could lead to incompatibilities between the host and target build. Comparable to other bugs, detection of incompatible software has a less significant impact should it be detected early on. In fact, building portable software is a merit on its own as software migration between target platforms improves code reuse.

Hardware dependencies

External dependencies, like hardware interaction, complicate the automation of tests. First, they need to be controlled to ensure deterministic execution of the tests. Furthermore hardware might not be available during software development. Regardless of the reason, in order to successfully program according to TDD, tests need to run frequently. This implies that executing tests should not depend on the target platform. Finally, in order to effectively use an external dependency in a test, setup and teardown will get considerably more complicated.

3.2 Test on Target

In the *Test on target* strategy, TDD issues raised by the target platform are not dealt with. Nevertheless, *Test on target* is a fundamental strategy as a means of verification. First, executing tests on target deliver feedback as part of an on-host development strategy. Moreover, during the development of system, integration or real-time tests, the effort in mocking specific hardware aspects is too labor intensive. Finally, writing validation tests when adopting TDD in a legacy code based system, provides a self-validating, unambiguous system to verify existing behavior.

3.2.1 Implementation

Fundamental to *Test on target* is a portable, low overhead test framework, as shown in Fig. 7. Secondary, some specific on-target test functionality, like timed asserts or

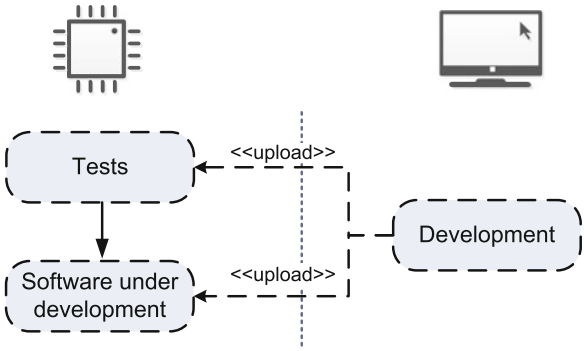


Fig. 7 *Test on target*

memory leak detection, are interesting features to include. Finally, it is important to consider the ease of adapting the framework when no standard output is available.

Tests are written in program memory of the target system, alongside the code under test itself. Generally, the test report is transmitted to the host system, to review the results of the test run. However, in extremely limited cases it becomes even possible to indicate passing or failing tests on a single LED. Yet, this implies that all free debug information is lost and therefore this should be considered as a final resort on very limited embedded systems.

3.3 Process

Test on target is too time-consuming to comfortably develop embedded software, because of frequent upload and execution cycles on target. Still it complements embedded TDD in three typical situations.

First, it extends the regular TDD cycle on host, in order to detect cross-platform issues, which is shown in the embedded TDD cycle, Fig. 8. Complementary to the TDD cycle on host, three additional steps are taken to discover incompatibilities between host and target. First, after tests are passing on the host system, the target compiler is invoked to statically detect compile-time errors. Next, once a day, if all compile-time errors are resolved, the automated on-target tests are executed. Finally, every few days, provided that all automated tests are passing, manual system or acceptance tests are done. Note that time indications are dependent on an equilibrium between finding cross-compilation issues early on and avoiding too much delays.

A second valuable use of *Test on target* is the development of target-dependent tests and code. For instance, memory management operations, real-time execution, on-target library functionality and IO-bound driver functions are impossible to test accurately on a host system. In these situations, forcing TDD on host will only delay development. Furthermore, an undesirable number of mock implementations are

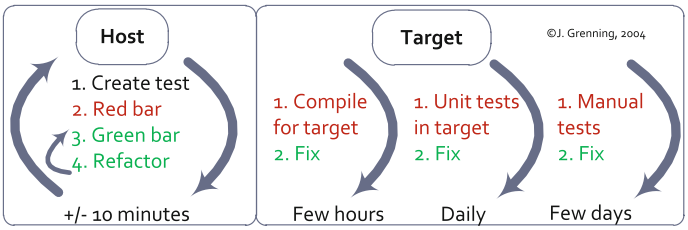


Fig. 8 Embedded TDD cycle

needed to solve some of these cases, resulting in tests that only test the mock. TDD should only be applied to software that is useful to test. When external software is encountered, minimize, isolate and consolidate its behavior.

Finally, *Test on target* has its merit to consolidate behavior in software systems without tests. Changing existing software without tests giving feedback on its behavior, is undesirable. After all this is the main reason to introduce TDD in the first place. However, chances are that legacy software does not have an accompanying test suite. Preceding refactoring of legacy software with on-target tests capturing the system's fundamental behavior is essential to safely conduct the necessary changes.

3.4 Code Example

In the following example, a focus is put on automation of tests for low level hardware-related software, according to the *Test on target* strategy.

```
TEST(RepeatButtonTest)
{
Button *button = new Button(&IOPIN0, 7);

button->setCurrentState(RELEASED);
CHECK(button->getState() == RELEASED);

button->setCurrentState(PRESSED);
CHECK(button->getState() == PRESSED);

button->setCurrentState(RELEASED);
button->setCurrentState(PRESSED);
CHECK(button->getState() == REPEAT);

delete button;
}
```

This test[1] creates a button object and tries to check whether its state logic functions correctly, namely two consecutive high states should result in REPEAT. Now, one way to test a button is to press it repeatedly and see what happens. Yet this requires manual interaction and is not feasible to manually test the button every time a code change is made. However, automation of events related to hardware can be solved with software. In this case an additional method is added to the button class, setCurrentState, which allows to press and release the button in software.

Two remarks are generally put forward when adding methods for testing purposes. On the one hand, these methods will litter production code. This can easily be solved by inheriting from the original class and add these methods in a test subclass. On the other hand, when a hardware event is mocked by some software, it might contain bugs on its own. Furthermore there is no guarantee that the mock software is a good representation of the hardware event it is replacing. Finally, is the actual code under test or rather the mock code tested this way?

These remarks indicate that manual testing is never ruled out entirely. In the case of automating tests and adding software for this purpose, a general rule of thumb is to test it both manually and automated. If both tests have identical results, consider the mock software as good as the original hardware behavior. The added value of the automated test will return on its investment when refactoring the code under test or extending its behavior.

3.5 Test on Host

Ideally, program code and tests reside in memory of the programmer's development computer. This situation guarantees the fastest feedback cycle in addition to independence of target availability. Furthermore, developing in isolation of target hardware improves modularity between application code and drivers. Finally, as the host system has virtually unlimited resources, a state of the art unit testing framework can be used.

In the *Test on host* strategy, development starts with tests and program code on the host system. However, calls to the effective hardware are irrelevant on the host system. Hence a piece of code replaces the hardware related functions, mocking the expected behavior. This is called a mock, i.e. a fake implementation is provided for testing purposes. A mock represents the developer's assumptions on hardware behavior. Once the developed code is migrated to the effective hardware system, these assumptions can be verified.

[1] All code snippets are based on the UnitTest++ framework.

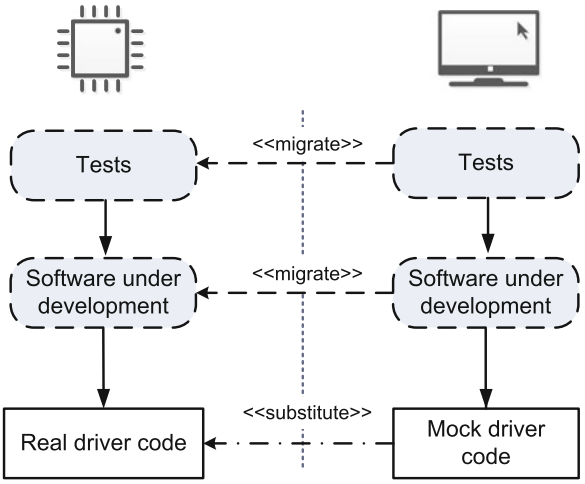


Fig. 9 *Test on host*

3.5.1 Implementation

The *Test on host* strategy typically consists of two build configurations, as shown in Fig. 9. Regardless of the level of abstraction of hardware, the underlying components can be mocked in the host build. This enables the developer to run tests on the host system, regardless of any dependency on the target platform. However, cross-platform issues might arise and these are impossible to detect when no reference build or deployment model is available. Ideally, building effectively for the real target platform will identify these issues. Although, running the cross-compiler or deploying to a development board could already identify some issues before the actual target is available.

3.5.2 Process

In order to deal with the slow cycle of uploading embedded program code, executing tests on target and reporting, *Test on host* is presented as the main solution. In order to do so an assumption is made that any algorithm can be developed and tested in isolation on the host platform. Isolation from hardware-related behavior is critical with the purpose of dynamically delegating the call to the real or mock implementation.

Considering the differences between host and target platform, verification of correspondence between target and host implementation is essential. These differences are:

- Cross-compilation issues, which occur as compilers can generate different machine code from the same source code. Also functions called from libraries

for each platform might lead to different results, as there is no guarantee regarding correspondence of both libraries.
- Assumptions on hardware-related behavior. Since the hardware reference is not available on host, the mocks are representing assumptions made on hardware behavior. This requires having an in-depth knowledge of hardware specifications. Furthermore, as the hardware platform for embedded systems can evolve, these specifications are not as solid or verified as is the case with a host system.
- Execution issues concerning the different platforms. These concern the difference in data representation, i.e. word-size, overflow, memory model, speed, memory access times, clocking differences, etc. These issues can only be uncovered when the tests are executed on the target platform.

Test on host is the primary step in the embedded TDD cycle [23, 27–29], as shown in Fig. 8. This cycle employs the technique of "dual targeting", which is a combination of *Test on host* and *Test on target*. In effect, development in this process is an activity entirely executed according to Test on host, as a reasonable development speed can be achieved. However, in order to cover up for the intrinsic deficiencies of *Test on host*, *Test on target* techniques are applied. Specifically, time-intensive activities are executed less frequent, which allows managing the process between development time and verification activities. The embedded TDD cycle proscribes to regularly compile with the target compiler and subsequently solve any cross-compilation issues. Next, automated tests can be ported to the target environment, execute them and solve any problems that arise. Yet, as this is a time-intensive activity it should be executed less frequently. Finally, some manual tests, which are the most labor-intensive, should only be carried out every couple of days.

3.5.3 Code Example

In this example, a one-wire reset method will be developed. As one-wire implies bidirectional communication on a single wire, the reset command will require that the connected pin direction changes in the process. This is the basis for the following test:

```
TEST(ResetTempSensorTest)
{
/* IO mapped memory representation on host */
unsigned int IOaddresses [8];

/* register to mock */
unsigned int *IODIRmock;

/* map it on the desired position in the array */
IODIRmock = IOaddresses + 7;
unsigned int pinNumber = 4;
```

```
/* mock external reset */
*IODIRmock = 0xFF;

TemperatureSensor *tempSensor =
new TemperatureSensor(IOaddresses, pinNumber);
tempSensor->reset();

/* test the change of direction */
CHECK_EQUAL(*IODIRmock,0xEF);
}
```

A problem is encountered since the test is going to run on host. Namely, memory-mapped IO registers are not available and addressing the same memory addresses on host will result in a crashing test. To deal with this problem, an array representing a contiguous chunk of memory can be used. This array has a twofold purpose. On the one hand it encourages the use of relative addressing, which isolates the use of hardware specific memory locations. On the other hand this array can be addressed by tests to simulate hardware behavior by changing the contents of the array directly. With constructor injection, the temperature sensor can be assigned to a specific pin on a port or when the test is executed on host, the mock can be injected that way.

This example is an illustration of how TDD influences code quality and low-level design and how *Test on host* amplifies this effect. As a prerequisite to *Test on host* hardware needs to be loosely coupled. This enables to reuse this code more easily, in case the temperature sensor is placed on a different pin or if the code needs to be migrated to a different platform.

3.6 Remote Testing

Test on host in conjunction with *Test on target* provides a complete development process, in order to successfully apply TDD. Yet, it introduces a significant overhead to maintain two separate builds and to write the hardware mocks. *Remote testing* [30] is an alternative, which eliminates both of these disadvantages (Fig. 10).

Remote testing is based on the principle that tests and code under test do not need to be implemented in the same environment. The main motivation to apply this to embedded software is the observation that TDD requires a significant number of uploads to the target system.

3.6.1 Implementation

Remote testing is based on the technology of remoting, for instance Remote Procedure Calls (RPC), Remote Method Invocation (RMI) or Common Object Request Broker Architecture (CORBA) [31]. Remoting allows executing subroutines

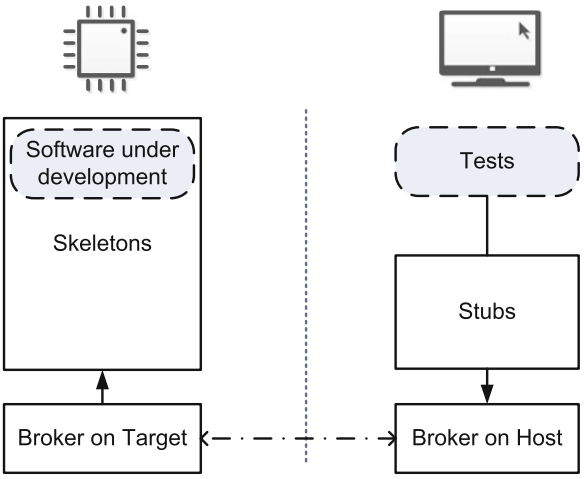


Fig. 10 *Remote testing*

in
another address space, without the manual intervention of the programmer. When this is applied to TDD for embedded software, remoting allows for tests on host to call the code under test, which is located on the target environment. Subsequently, the results of the subroutine on target are returned to the *Test on host* for evaluation.

Regardless of the specific technology, a broker is required which will setup the necessary infrastructure to support remoting. In homogeneous systems, such as networked computing, the broker on either side is the same. However, because of the specific nature of embedded systems, a fundamental platform difference between the target and the host broker exists.

On the one hand the broker on target has a threefold function. First, it maintains communication between host and target platform on the target side. Next, it contains a list of available subroutines which are remotely addressable. Finally, it keeps a list of references to memory chunks, which were remotely created or are remotely accessible. These chunks are also called skeletons.

On the other hand the broker on host serves a similar, but slightly different function. For one thing it maintains the communication with the target. Also, it tracks the stubs on host, which are interfaces on host corresponding to the skeletons in the target environment. These stubs provide an addressable interface for tests, as if the effective subroutine would be available in the host system. Rather than executing the called function's implementation, a stub merely redirects the call to the target and delivers a return value as should the function have been called locally.

As the testing framework solely exists in the host environment, there is practically no limitation on it. Even the programming language on host can differ completely from the target's programming language. In the spirit of CxxTest[2] a note is made

[2] CxxTest [32] is a C++ testing framework, which was written in Python.

that C++ as a programming language to devise tests might require a larger amount of boilerplate code than strictly necessary. Writing tests in another language is a convenience which can be exploited with *Remote testing*.

Unfortunately, the use of remoting technology introduces an overhead into software development. Setting up broker infrastructure and ensuring subroutines are remotely accessible require a couple of additional actions. On target, a subroutine must be extended to support a remote invocation mechanism called marshaling. This mechanism will allow the broker to invoke the subroutine when a call from host "marshals" such an action. Correspondingly on host, an interface must be written which is effectively identical to the interface on target. Invoking the subroutine on host will marshal the request, thus triggering the subroutine on target, barring communication issues between host and target.

Some remoting technologies, for instance CORBA, incorporate the use of an Interface Description Language (IDL). An IDL allows defining an interface in a language neutral manner to bridge the gap between otherwise incompatible platforms. On its own the IDL does not provide added value to remoting. However the specifications describing the interfaces are typically used to automatically generate the correct serialization format. Such a format is used between brokers to manage data and calls. As serialization issues concern the low level mechanics of remoting, an IDL provides a high level format, which relieves some burden of the programmer.

3.6.2 Process

The *Remote testing* development cycle changes the conventional TDD cycle in the first step. When creating a test, the interface of the called subroutine under test must be remotely defined. This results in the creation of a stub on host which makes the defined interface available on the host platform, while the corresponding skeleton on target must also be created. Subsequent steps are straightforward, following the traditional TDD cycle.

1. Create a test
2. Define an interface on host
   (a) Call the subroutine with test values
   (b) Assert the outcome
   (c) Make it compile
   (d) If the subroutine is newly created: add a corresponding skeleton on target
   (e) Run the test, which should result in a failing test
3. Red bar
   (a) Add an implementation to the target code
   (b) Flash to the target
   (c) Run the test, which should result in a passing test
4. Green bar: either refactor or add a new test.

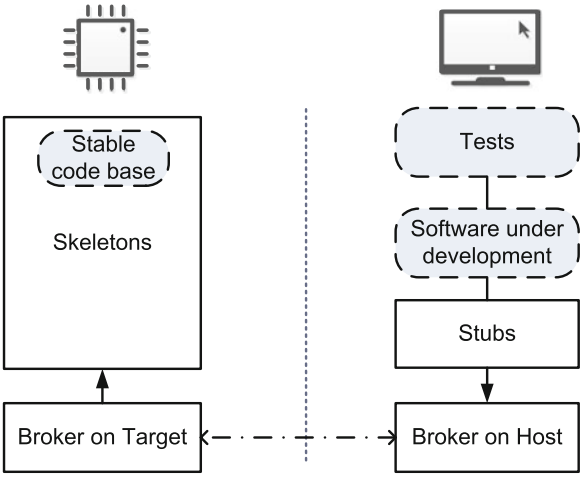


Fig. 11 Remote prototyping

Remote testing only provides a means to eliminate one code upload. In order to deal with the rather low return of investment inherent to *Remote testing*, an adaption to the process is made, which results in a new process called *Remote prototyping*.

3.6.3 Remote Prototyping

Principally *Remote prototyping* involves developing code on the host platform, while specific hardware calls can be delegated towards the respective code on target [33]. Addressing the hardware-related subroutines on host, delegating the call to the target and returning values as provided by the subroutine prototype are provided by remoting infrastructure.

Software can be developed on host, as illustrated in Fig. 11, as all hardware functionality is provided in the form of subroutine stubs. These stubs deliver the subroutine definition on the host system while an effective call to the subroutine stub will delegate the call to the target implementation.

Remote prototyping is viable under the assumption that software under development is evolving, but once the software has been thoroughly tested, a stable state is reached. As soon as this is the case the code base can be instrumented to be remotely addressable. Subsequently, it is programmed into the target system and thoroughly tested again to detect cross-compilation issues. Once these issues have been solved, the new code on target can be remotely addressed with the aim of continuing development on the host system.

An overview of the *Remote prototyping* process applied to an object oriented implementation, for instance C++, is given in Fig. 12. A fundamental difference

exists when all objects can be statically allocated or whether dynamic creation of memory objects is required.

In a configuration in which the target environment can be statically created, setup of the target system can be executed at compile time. The broker system is not involved in constructing the required objects, yet keeps a reference to the statically created objects. Effectively the host system does not need to configure the target system and treats it as a black box. Conversely the process of *Remote prototyping* with dynamic allocation requires additional configuration. Therefore the target system is approached as a glass box system. This incurs an additional overhead for managing the on target components, yet allows dynamically reconfiguring the target system without wasting a program upload cycle.

The dynamical *Remote prototyping* strategy starts with initializing both the broker as well on target as on host side. Next, a test is executed, which initializes the environment. This involves setting up the desired initial state on the target environment. This is in anticipation of the calls, which the software under development will conduct. For instance, to create an object in the target, the following steps are performed, as illustrated in Fig. 12.

1. The test will call the stub constructor, which provides the same interface as the actual class.
2. The stub delegates the call to the broker on host.
3. The broker on host translates the constructor call in a platform independent command and transmits it to the target broker.
4. The broker on target interprets the command and calls the constructor of the respective skeleton and in the meanwhile assigns an ID to the skeleton reference.
5. This ID is transmitted in an acknowledge message to the broker on host, which assigns the ID to the stub object.

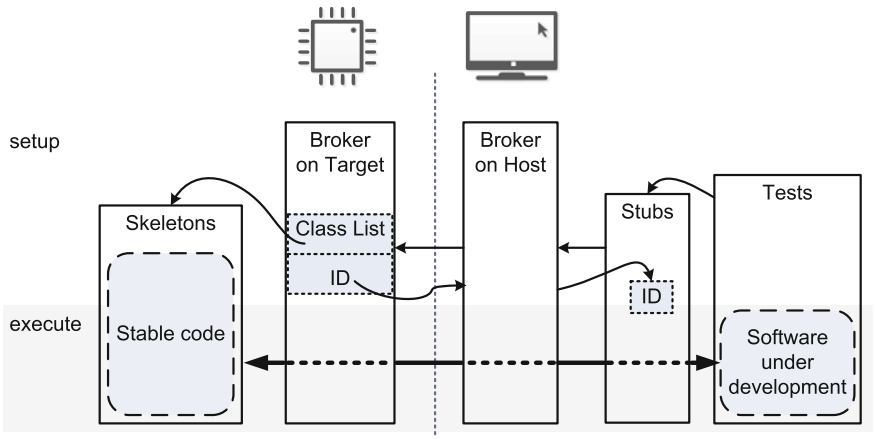


Fig. 12 *Remote prototyping* process with dynamically allocated objects

After test setup, the test is effectively executed. Any calls to the hardware are dealt with by the stub object, which are delegated to the effective code on target. Likewise, any return values are delivered to the stub. Optionally another test run can be done without rebooting the target system. A cleanup phase is in order after each test has executed, otherwise the embedded system would eventually run out of memory. Deleting objects on target is as transparent as on host, with the addition that the stub must be cleaned up as well.

Remote prototyping deals with certain constraints inherent to embedded systems. However, some issues can be encountered when implementing and using the Remoting infrastructure.

Embedded constraints

The impact, especially considering constrained memory footprint and processing power, of the remoting infrastructure on the embedded system is minimal. Of course it introduces some overhead to systems which do not need to incorporate the infrastructure for application needs. On the other hand *Remote prototyping* enables conducting unit tests with a real target reference. Porting a unit test framework and running the tests in target memory as an alternative will introduce a larger overhead than the remoting infrastructure and lead to unacceptable delays in an iterative development process.

Next, the embedded infrastructure does not always provide all conventional communication peripherals, for instance Ethernet, which could limit *Remote prototyping* applicability. However, if an IDL is used, the effective communication layer is abstracted. Moreover, the minimal specifications needed to setup *Remote prototyping* are limited as throughput is small and no timing constraints need to be met.

Finally, *Remote prototyping* requires that hardware and a minimalistic hardware interfacing is available. This could be an issue when hardware still needs to be developed. Furthermore hardware could be unavailable or deploying code still under development might be potentially dangerous. Lastly, a minimalistic software interface wrapping hardware interaction and implementing the remoting infrastructure is needed to enable remote prototyping. This implies that it is impossible to develop all firmware according to this principle.

Issues

The encountered issues when implementing and using *Remote prototyping* can be classified in three types. First are cross-platform issues related to the heterogeneous architecture. A second concern arises when dynamic memory allocation on the target side is considered. Thirdly, translation of function calls to common architectural independent commands introduces additional issues.

Differences between host and target platform can lead to erratic behavior, such as unexpected overflows or data misrepresentation. However, most test cases will quickly detect any data misrepresentation issues. Likewise, over- and underflow problems can be discovered by introducing some boundary condition tests.

Next, on-target memory management is an additional consideration which is a side-effect of *Remote prototyping*. Considering the limited memory available on

target and the single instantiation of most driver components, dynamic memory allocation is not desired in embedded software. Yet, *Remote prototyping* requires dynamic memory allocation to allow flexible usage of the target system. This introduces the responsibility to manage memory, namely creation, deletion and avoiding fragmentation. By all means this only affects the development process and unit verification of the system, as in production this flexibility is no longer required.

Finally, timing information between target and host is lost because of the asynchronous communication system, which can be troublesome when dealing with a real-time application. Furthermore to unburden the communication channel, exchanging simple data types are preferred over serializing complex data.

Tests

The purpose of *Remote prototyping* is to introduce a fast feedback cycle in the development of embedded software. Introducing tests can identify execution differences between the host and target platform. In order to do so the code under test needs to be ported from the host system to the target system. By instrumenting code under test, the *Remote prototyping* infrastructure can be reused to execute the tests on host, while delegating the effective calls to the code on target.

3.6.4 Overview

Test on target, *Test on host*, *Remote testing* and *Remote prototyping* have been defined as strategies to develop in a TDD fashion for embedded. These strategies have advantages and disadvantages when a comparison between them is made [34]. Furthermore, because of the disadvantages, these strategies excel in a particular embedded environment. In this section a comparison is made between each strategy and an overview is given of how development in a project can be composed of combinations of these strategies.

The baseline of this comparison is *Test on target*. Namely for the particular reason that when the number of code uploads to target is the only consideration, *Test on target* is the worst strategy to choose. It is possible to demonstrate this when the classical TDD cycle is considered, as in Fig. 13.

When TDD is strictly applied in *Test on target*, every step will require a code upload to target. Considering the iterative nature of TDD, each step will be frequently

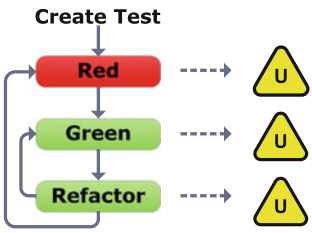


Fig. 13 Uploads to target when *Test on target* is considered

run through. Moreover since a target upload is a very slow act, this will deteriorate the development cycle to a grinding halt. Thus reducing the number of uploads is critical in order to successfully apply TDD for embedded software.

Remote Testing

When considering the effect of *Remote testing* on TDD for embedded software, the following observation can be made. At a minimum with *Test on target*, each test will require two uploads, i.e. one to prove the test is effectively failing and a second one, which contains the implementation to make the test pass. Note that this is under the assumption that the correct code is immediately implemented and no refactorings are needed. If it takes multiple tries to find the correct implementation or when refactoring the number of uploads rises.

In order to decrease the number of required uploads, tests can be implemented in the host environment, i.e. *Remote testing*. Effectively this reduces the number of uploads by the number of tests per subroutine minus one. One is subtracted because a new subroutine will require to flash the empty skeleton to the target. Therefore the benefit of *Remote testing* as a way to apply TDD to embedded software is limited, as demonstrated in Table 1. The ideal case is when a test passes after the first implementation is tried.

Consider that tests have a relative low complexity when compared to production code. This observation implies that a test is less likely to change than the effective code under test, which indicates a reduction of the benefits of *Remote testing*. The possibility of changing code to reach green bar, namely during the implementation or refactoring phase, is higher than the (re)definition of the tests. Effectively this will reduce the ratio of tests versus code under test requiring an update of code on target. When only the number of uploads is taken into account, *Remote testing* will never be harmful to the development process. Yet considering the higher complexity of production code and refactoring, which mostly involves changing code under test, the benefit of *Remote testing* diminishes rapidly. When other costs are taken into account, this strategy is suboptimal when compared to Test on host. However, as a pure testing strategy, *Remote testing* might have its merit. Though the application of *Remote testing* in this context was not further explored.

Table 1 *Remote testing* benefit

	# Tests : # code uploads : # new remote subroutines	Remote testing (%)
Worst case	1 : 1 : 1	0
2 TDD cycles (ideal)	2 : 2 : 1	25
3 TDD cycles (ideal)	3 : 3 : 1	33
X TDD cycles (ideal)	X : X : 1	Max = 49.99...
General case	T : C : R	$\frac{T-R}{T+C} * 100$

Remote Prototyping

As the effectiveness of *Remote testing* is limited, an improvement to the process is made when code is also developed on host, i.e. Remote prototyping. *Remote prototyping* only requires a limited number of remote addressable subroutines to start with. Furthermore, once code under development is stable, its public subroutines can be ported and made remote addressable in turn. This is typically when an attempt can be made to integrate newly developed code into the target system. At that moment these subroutines can be addressed by new code on host, which is of course developed according to the *Remote prototyping* principle.

Where *Remote prototyping* is concerned, it is possible to imagine a situation which is in fact in complete accordance to *Remote testing*. Namely when a new remote subroutine is added on target, this will conform to the idea of executing a *Test on host*, while code under test resides on target. However code which is developed on host will reduce the number of uploads, which would normally be expected in a typical *Test on target* fashion. Namely each action which would otherwise have provoked an additional upload will add to the obtained benefit of *Remote prototyping*.

Yet the question remains of how the *Remote testing* part of the process is related to the added benefit of *Remote prototyping*. A simple model to express their relation is the relative size of the related source code. Namely on the one hand the number of Lines Of Code (LOC) related to remoting infrastructure added to the LOC of subroutines which were not developed according to *Remote prototyping*, but rather with Remote testing. On the other hand the LOC of code and tests developed on host. These assumptions will ultimately lead to Table 2.

In Table 2 the following symbols are used:

T	# tests
C	# of code uploads
R	# of new remote subroutines
C_D	# of uploads related to *Remote testing*
C_H	# of averted uploads on host
LOC_D	LOC related to *Remote testing* (Remoting infrastructure + traditionally developed code)
LOC_H	LOC developed according to *Remote prototyping*

$$\alpha = \frac{LOC_D}{LOC_{TOTAL}}, \beta = \frac{LOC_H}{LOC_{TOTAL}}$$

Table 2 indicates a net improvement of *Remote prototyping* when compared with *Remote testing*. Furthermore it also guarantees an improvement when compared to *Test on target*. Nevertheless it also shows the necessity of developing code and tests on host as the major benefit is obtained when β and C_H are high when they are compared with respectively α and T.

Table 2 *Remote prototyping* benefit

	# Tests	α	Target (C_D : R)	β	Host (C_H)	Remote prototyping (%)
	1	0	/	1	1	50
	1	0	/	1	2	66
Max.	1	0	/	1	Y	Max = 99.99...
	1	0.75	1 : 1	0.25	1	12.5
Min.	1	$1-\beta$	1 : 1	β	1	Min = $\frac{1}{2}\beta * 100$
General case	T	α	C_D : R	β	C_H	$\left(\alpha \frac{T-R}{T+C_D} + \beta \frac{C_H}{T+C_H}\right) * 100$

Test on Host

Finally, a comparison can be made with the *Test on host* strategy. When only uploads to target are considered, *Test on host* provides the best theoretical maximum performance, as it only requires one upload to the target, i.e. the final one. Ofcourse, this is not a realistic practice and definitely contradicts with the incremental aspect of TDD. Typically a verification upload to target is a recurrent irregular task, executed at the discretion of the programmer. Furthermore *Test on host* and the remoting strategies have another fundamental difference. Namely while setting up remoting infrastructure is only necessary when a certain subroutine needs to be remotely addressable, *Test on host* requires a mock. Although there are mocking frameworks which reduce the burden of manually writing mocks, it still requires at least some manual adaptation. When the effort to develop and maintain mocks is ignored, a mathematical expression similar to the previous expressions can be composed as shown in Table 3. However it should be noted, that this expression does not consider the most important metric for *Test on host* and is therefore less relevant[3]

In Comparison

In the previous sections, the only metric which was considered was the number of code uploads. Although this is an important metric to determine which strategy is more effective, there are also other metrics to consider.

First metric is the limited resources of the target system, namely memory footprint and processing power. On the one hand when considering the *Test on target* case, tests

Table 3 *Test on host* benefit when only target uploads are considered

	# Tests (T)	# Code uploads (C)	# Verification uploads (U)	Test on host (%)
	1	1	1	50
Min.	1	C	U	Min = 0.0...1
Max.	T	C	1	Max = 99.99...
General case	T	C	U	$\left(1 - \frac{U}{T+C}\right) * 100$

[3] This expression is included nevertheless, for the sake of completeness.

Table 4 *Test on target*, *Test on host* and *Remote prototyping* in comparison

	Test on target	Test on host	Remote prototyping
Slow upload	− − −	+ + +	++
	Test and program on target	Test and program on host	Broker on target
Restricted resources	− − −	+ + +	+/−
	Target memory and processing power	Host memory and processing power	Host memory and target processing power
Hardware dependencies	+ + +	− − −	+/−
	Real hardware	Mock hardware	Intermediate format
Overhead	+	− − −	−−
	Test framework	Mocks	Remoting infrastructure

and a testing framework will add to the required memory footprint of the program. While on the other hand, the processing power of the target system is also limited, so a great number of tests on target will slow down the execution of the test suite. Another metric to consider are the hardware dependencies, namely how much effort does it require to write tests (and mocks) for hardware-related code? Finally, what is the development overhead required to enable each strategy. For *Test on target* this is the porting of the testing framework, while *Test on host* requires the development and maintenance of hardware mocks and finally *Remote prototyping* requires Remoting infrastructure.

Table 4 provides a qualitative overview of the three strategies compared to each other when these four metrics are considered.

The overview in Table 4 does not specify the embedded system properties, as the range of embedded systems is too extensive to include this information into a decision matrix. For instance, applying *Remote prototyping* does not have any overhead at all, when remoting infrastructure is already available. Likewise when an application is developed on embedded Linux, one can develop the application on a PC Linux system with only minimal mocking needed, making *Test on host* the ideal choice. Moreover in this overview no consideration is given to legacy code, yet the incorporation of legacy code will prohibit the use of the *Test on host* strategy.

When deciding which strategy is preferable, no definite answer can be given. In general, *Test on target* is less preferred than Test on host and *Remote prototyping*, while *Remote prototyping* is strictly better than *Remote testing*. Yet beyond these statements all comparisons are case-specific. For instance when comparing *Test on host* versus *Remote prototyping*, it is impossible to make a sound decision without considering the embedded target system and the availability of drivers, application software, etc.

4 Embedded TDD Patterns

The following sections deal with two structural patterns related to Test-Driven Development for embedded software. First is 3-tier TDD, which deals with the different levels op complexity to develop embedded software. Subsequently is the MCH pattern [35–37], an alternative with the same objective.

4.1 3-Tier TDD

In dealing with TDD for embedded software, three levels of difficulty to develop according to TDD are distinguished. Each of these levels imply their specific problems with each TDD4ES strategy. A general distinction is made between hardware-specific, hardware-aware and hardware independent code. When developing these types of code, it is noted that the difficulty to apply TDD increases, as shown in Fig. 14.

Hardware independent code

A fine example of hardware independent software is code to compose and traverse data structures, state machine logic, etc. Regardless of the level of abstraction these parts of code could be principally reused on any platform. Typically hardware independent code is the easiest to develop with TDD. Because barring some cross-compilation issues, it can be tested and developed on host (either as *Test on host* or *Remote prototyping*). Typical issues that arise when porting hardware independent code to target are:

- Type-size related, like rounding errors or unexpected overflows. Although most of these problems are typically dealt with by redefining all types to a common size

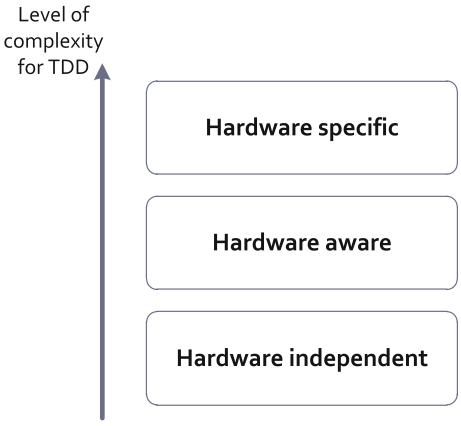


Fig. 14 Three tiers of TDD for embedded software

across the platforms, it should still be noted that unit tests on host will not detect any anomalies. This is the reason to run tests for hardware independent code.
- Associated with the execution environment. For instance, execution on target might miss deadlines or perform strangely after an incorrect context switch. The target environment is not likely to have the same operating system, provided it has an OS, as the host environment.
- Differences in compiler optimizations. Compilers might have a different effect on the same code, especially when optimizations are considered. Also problems with the volatile keyword can be considered in this category. Running the compiler on host with low and high optimization might catch some additional errors.

Most importantly, developing hardware independent code according to TDD requires no additional considerations for each of the strategies. Concerning *Test on target*, the remark remains that all development according to this strategy is painstakingly slow. Furthermore hardware independent code does not impose any limitations on either Remote prototyping or *Test on host*, so there should be no reason to develop according to *Test on target*.

Hardware-aware code

Next is hardware-aware code, which is a high level abstraction of target-specific hardware components. A typical example of a hardware-aware driver is a Temperature sensor module. This does not specify which kind of Temperature sensor is used. It might as well concern a digital or analog temperature sensor. Yet it would not be surprising to expect some sort of getTemperature subroutine. Hardware-aware code will typically offer a high level interface to a hardware component, yet it only presents an abstraction of the component itself, which allows changing the underlying implementation. Developing hardware-aware code on host will require a small investment when compared to a traditional development method, because hardware-aware code will typically call a low level driver, which is not available on host. However the benefits of TDD compensate for this investment. The particular investment depends on the strategy used.

On the one hand when developing according to *Test on host* this investment will be a mock low level driver. The complexity of this mock depends on the expected behavior of the driver. This particular approach has two distinct advantages. First, it allows intercepting expected non-deterministic behavior of the driver, which would otherwise complicate the test.

A second advantage of using mocks to isolate hardware-aware code for testing purposes. Namely, a consequence of the three-tier architecture is that unit tests for hardware-aware code will typically test from the hardware independent tier. This has two reasons. On the one hand a unit test typically approaches the unit under test as a black box. On the other hand, implementation details of hardware-aware and hardware-specific code are encapsulated, which means only the public interface is available for testing purposes. In order to deal with unit test limitations, breaking encapsulation for testing is not considered as an option. Because it is not only considered as a harmful practice, but is also superfluous as mocks enable testing the

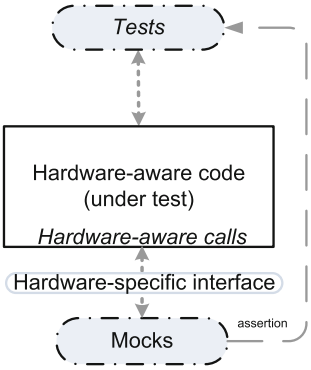


Fig. 15 Isolating the hardware-aware tier with a mock hardware component

man-in-the-middle, also known as hardware-aware code. An overview of this pattern is shown in Fig. 15.

Fundamental to the idea of using mocks is that the actual assertion logic is provided in the mock instead of in the test. Whereas tests can test the publicly available subroutines, a mock, which is put into the lower tier can assert some internals of the code under test, which would otherwise be inaccessible.

The other preferred approach to develop hardware-aware code is Remote prototyping. As a strategy *Remote prototyping* is optimized to deal with hardware-aware code. Namely, developing a part of code which is loosely coupled to a hardware driver only requires the hardware driver functions to be remotely addressable. Once this condition is fulfilled it enables developing the rest of the code on host, without the need of developing mocks.

Yet when considering testing hardware-aware code as a black box, the addition of mocks allowed to test from a bottom-up approach. As Remote prototyping does not require including mocks, it appears to be limited to the typical top-down testing style. To make it worse, injecting mocks with *Remote prototyping* is a convoluted process, which is not recommended.

Nevertheless mocks, or at least similar functionality, can be introduced in a typical *Remote prototyping* process. Instead of injecting a mock, the respective stub can be enhanced with the aforementioned assertion logic. This creates a mock/stub hybrid, which one the one hand delegates calls to target and on the other hand records and validates the calls from the code under test. Figure 16 presents this mock/stub hybrid in the context of *Remote prototyping* hardware-aware code.

A mock/stub hybrid allows to execute a double assertion, namely whether the value is correct and the assertion logic provided by the mock part. Furthermore it can be extended with more advanced mocking capabilities, like raising events, returning fixed data, etc. This counteracts the principle of *Remote prototyping* of calling the actual code on target, but allows introducing determinism in an otherwise non-deterministic process. For instance, to explore the failure path of a sensor reading in a deterministic way it would be too time-consuming to execute actual sensor

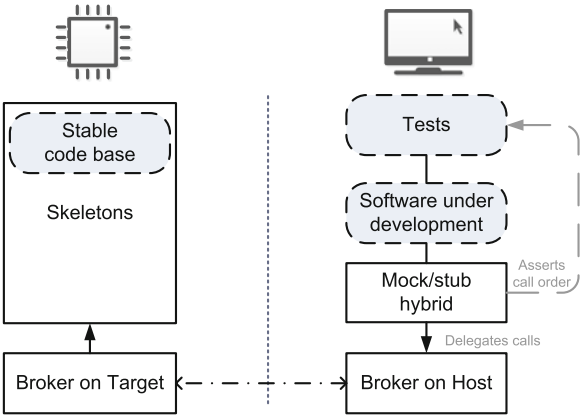


Fig. 16 *Remote prototyping* with a mock/stub hybrid, which can assert the call order of the software under test

readings until a failure has been invoked. Therefore it is easier to mock the failure, which guarantees a failure every time the test is executed.

Hardware-specific code

Finally hardware-specific code is the code which interacts with the target-specific registers. It is low level driver code, which is dependent on the platform, namely register size, endianness, addressing the specific ports, etc. It fetches and stores data from the registers and delivers or receives data in a human-readable type, for instance string or int. An example of a hardware-specific code are drivers for the various peripherals embedded in a microcontroller.

Hardware-specific code is the most difficult to develop with TDD, as test automation of code which is convoluted with hardware is not easily done. When considering the strategies *Test on host* and Remote prototyping, each of these has its specific issues. On the one hand, *Test on host* relies on mocks to obtain hardware abstraction. Although it can be accomplished for hardware-specific code, as demonstrated in listing Sect. 3.5.3, developing strictly according to this strategy can be a very time absorbing activity. This would lead to a diminishing return of investment and could downright turn into a loss when compared to traditional development methods. Furthermore as hardware-specific code is the least portable, setting up tests with special directives for either platform could be an answer. However these usually litter the code and are only a suboptimal solution.

Optimally, the amount of hardware-specific code is reduced to a minimum and isolated as much as possible to be called by hardware-aware code. The main idea concerning hardware-specific code development is to develop low-level drivers with a traditional method and test this code afterwards. For both *Test on host* and *Remote prototyping* this results in a different development cycle.

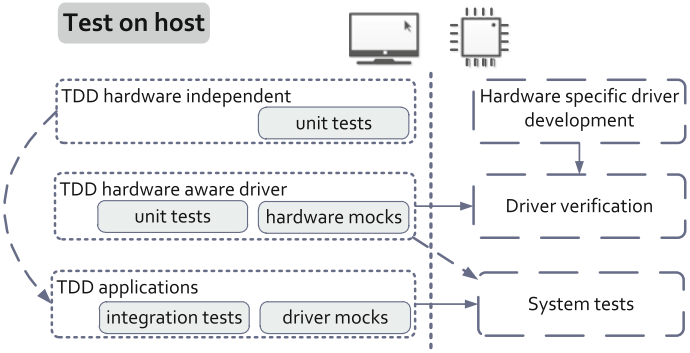


Fig. 17 3-tier development process with *Test on host*

3-tier TDD and *Test on host*

Test on host allows the concurrent development of hardware independent and specific code, as shown in Fig. 17. As previously indicated, hardware independent code naturally lends to test first development, while hardware-specific driver code can be tested afterwards. Once a minimal interface and implementation of hardware-specific code is available, hardware-aware code development can be started. Hardware mocks resembling hardware-specific behavior are used to decouple the code from target and enable running the tests on host. Once both hardware-specific and hardware-aware code of the driver has reached a stable state, the hardware-aware part can be migrated to target and instead of the mock the real hardware-specific code can be used. At that moment hardware independent code can be integrated with mocks which provide the hardware-aware interface. Finally all code can be combined on the target system, to perform the system tests.

3-tier TDD and *Remote prototyping*

On the other hand the *Remote prototyping* process starts the same, but differs from *Test on host* in the later steps. An attempt could be made to mock hardware internals to enable hardware-specific development with *Test on host*. However, this is not possible for *Remote prototyping* as a minimal remote addressable function must be available on target. This naturally leads to a more traditional fashion of hardware-specific code development, yet testing afterwards might as well be according to the principles of *Remote testing*.

Nevertheless, once a minimal function is available it becomes possible to develop using the *Remote prototyping* strategy. As indicated in Fig. 18, hardware-aware driver development uses remote calls to the hardware-specific drivers. The infrastructure for these calls is already present, should the hardware-specific drivers have been tested with *Remote testing*. Once a stable state has been reached a migration of hardware-aware code is in order to be incorporated in the remote addressable target system. Finally, the applications can be developed in the same fashion. A final migration allows performing system tests on target.

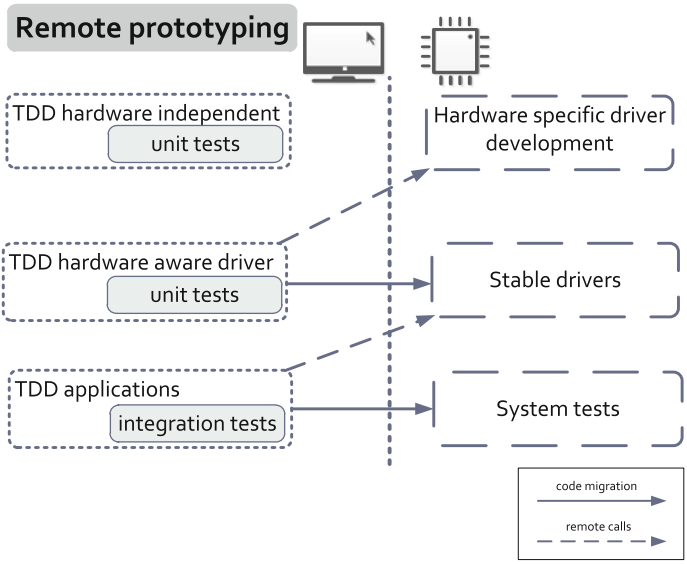


Fig. 18 3-tier development process with *Remote prototyping*

4.2 MCH-Pattern

An alternative for 3-tier TDD is the MCH-pattern by Karlesky et al., which is shown in Fig. 19. This pattern is a translation of the MVC pattern [38] to embedded software. It consists of a Model, which presents the internal state of hardware. Next is the Hardware, which presents the drivers. Finally the Conductor contains the control logic, which gets or sets the state of the Model and sends command or receives triggers from the Hardware. As this system is decoupled it is possible to replace each component with a mock for testing purposes.

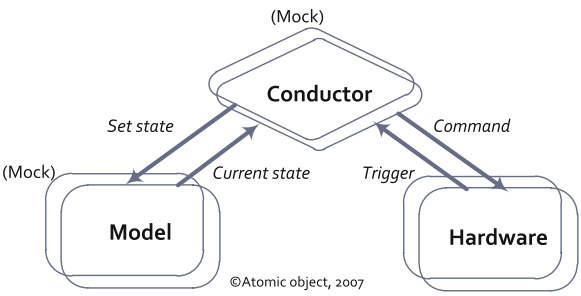


Fig. 19 MCH pattern

5 Conclusion

Test-Driven Development has proven to be a viable alternative to traditional development, even for embedded software. Yet a number of considerations have to be made. Most importantly, TDD is a fast cycle, yet embedded software uploads are inherently slow. To deal with this, as shown in the strategies, it is of fundamental importance to develop as much as possible on host. Therefore *Remote prototyping* or *Test on host* is preferred. Choosing between the former and the latter is entirely dependent on the target embedded system, tool availability and personal preference. Once the overhead of one of these strategies could be greatly reduced the balance may shift in favor of one or the other. Yet, at the moment of writing, *Test on host* is the most popular. However *Remote prototyping* might present a worthy alternative.

Besides *Remote testing* and *prototyping*, the main contribution of this manual and the research it describes is 3-tier TDD. This pattern allows isolating hardware and non-deterministic behavior, which are both prerequisites for test automation. This pattern presents a main guideline, which is not only applicable to development with TDD, but generally relevant for all embedded software development. Namely, minimizing the hardware-specific layer improves a modular design, loosely coupled to the hardware system. Such a design is more testable, thus its quality can be assured. Furthermore the software components could be reused over different hardware platforms. This is not only a benefit in the long run, when hardware platform upgrades are to be expected. Moreover, it will help the hardware and software integration phase. In this phase unexpected differences in hardware specifications can be more easily solved in changing the software component. Automated test suites will ensure that changing hardware-specific code to fit the integration does not break any higher-tier functionality. Or at least it will be detected by a red bar.

Future directions

1. Hardware mocking
   As briefly indicated in Sect. 2 mocks could be partially automatically generated by a mocking framework, which is complementary to a testing framework. No further elaboration is given on the subject, but since hardware mocks are extensively used in the *Test on host* strategy, a part of the work could be lifted from the programmer.
2. Related development strategies
   Test-Driven Development is a fundamental practice in Agile or eXtreme Programming methodologies. Yet, similar practices exist based on the same principles of early testing. For instance, Behavior-Driven Development (BDD) is an iterative practice where customers can define features in the form of executable scenarios. These scenarios are coupled to the implementation. In turn this can be executed indicating whether the desired functionality has been implemented. BDD for embedded has some very specific issues, since functionality or features in embedded systems is mostly a combination of hardware and software.

3. Code instrumentation for the Remote strategies
   Developing stubs and skeletons for the Remote strategies requires a considerable effort. However, the boilerplate code which is needed for the remoting infrastructure, could be generated automatically once the interface has been defined on host.
4. Quantitative evaluation of development strategies
   In an effort to compare the development strategies a qualitative evaluation model has been developed (Sect. 3.6.4). This model allows conducting quantitative case studies in a uniform and standardized manner. Since the model is a simplified representation of the actual process, it must be validated first. For instance by a number of quantitative case studies, it could be indicated that the model is correct. These would also allow further refinement of the model, so it incorporates additional parameters. Finally an attempt could be made to generalize the models to the development of other types of software.
5. Testing
   TDD is strongly involved with testing, however as a testing strategy it does not suffice. Real-time execution of embedded software is in some cases a fundamental property of the embedded system. However it is impossible to test this feature while developing, as premature optimization leads to a degenerative development process. Nevertheless another practice from Agile is fundamental in the development process. Continuous Integration (CI) is a process which advocates building a working system as much as possible. Running a suite of automated unit, integration and acceptance tests overnight indicates potential problems. Adding real-time specification tests in a nightly build might be able to detect some issues. However, considering the case of premature optimization, a certain reservation towards the value of these tests on a low level must be regarded.
   Testing concurrent software is another issue which cannot be covered by tests devised in a TDD process. As multi-core processors are getting incorporated in embedded systems, these issues will become more important.

References

1. B.W. Boehm, *Software Engineering Economics (Prentice-Hall Advances in Computing Science and Technology Series)* (Prentice Hall PTR, 1981)
2. K. Beck, *Test-Driven Development: By Example* (Addison-Wesley, 2003)
3. M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999)
4. R. Martin, *Clean Code: A Handbook of Agile Software Craftmanship* (Prentice Hall, 2008)
5. B. George, L. Williams, A structured experiment of test-driven development. J. Inf. Softw. Technol. Elsevier (2004)
6. M. Siniaalto, Test driven development: empirical body of evidence. Technical report, ITEA (2006)
7. N. Nagappan, M. Maximilien, T. Bhat, L. Williams, Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Softw. Eng. **13**, 289–302 (2008)
8. M. Müller, F. Padberg, About the return on investment of test-driven development, in *International Workshop on Economics-Driven Software Engineering Research EDSER-4*, 2003

9. N. Van Schooenderwoert, Embedded extreme programming: an experience report, in *Embedded Systems Conference (ESC), Boston*, 2004
10. N. Van Schooenderwoert, Embedded agile: a case study in numbers, in *Embedded Systems Conference (ESC), Boston*, 2006
11. B. Greene, Using agile testing methods to validate firmware. Agile Alliance Newsl. **4**, 79–81 (2004)
12. P. Hamill, *Unit Test Frameworks* (O'Reilly, 2004)
13. N. Llopis, Games from within: exploring the c++ unit testing framework jungle, http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle
14. Jtn002-minunit-a minimal unit testing framework for c, http://www.jera.com/techinfo/jtns/jtn002.html
15. Embedded unit testing framework for embedded c, http://embunit.sourceforge.net/embunit/ch01.html
16. Unity-test framework for c, http://sourceforge.net/apps/trac/unity/wiki
17. N. Llopis, C. Nicholson, Unittest++, http://unittest-cpp.sourceforge.net/
18. Cpputest, http://www.cpputest.org/
19. Googletest, google c++ testing framework, http://code.google.com/p/googletest/
20. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley, 2007)
21. M. Fowler, Inversion of control containers and the dependency injection pattern (2004), http://martinfowler.com/articles/injection.html
22. R. Koss, J. Langr, Test driven development in c. C/C++ Users J. (2002)
23. J. Grenning, *Test-Driven Development for Embedded C* (The Pragmatic Bookshelf, 2011)
24. J. Boydens, P. Cordemans, E. Steegmans, Test-driven development of embedded software, in *Proceedings of the Fourth European Conference on the Use of Modern Information and Communication Technologies*, 2010
25. P. Cordemans, S. Van Landschoot, J. Boydens, Migrating from debugging to testing embedded software, in *Proceedings of the 9th International Conference and Workshop on Ambient Intelligence and Embedded Systems (AmiEs)*, 2010
26. P. Cordemans, S. Van Landschoot, J. Boydens, Test-driven development in the embedded world, in *Proceedings of the First Belgium Testing Days Conference*, 2011
27. J. Grenning, Test-driven development for embedded c++ programmers. Technical report, Renaisance Software Consulting, 2002
28. J. Grenning, Progress before hardware. Agile Alliance Newsl. **4**, 74–79 (2004)
29. J. Grenning, Test driven development for embedded software, in *Proceedings of the Embedded Systems Conference 241*, 2007
30. Host / target testing with the ldra tool suite, http://www.ldra.com/host_trg.asp
31. Corba/e: industry-standard middleware for distributed real-time and embedded computing (2008), http://www.corba.org/corba-e/corba-e_flyer_v2.pdf
32. Cxxtest, http://cxxtest.tigris.org/
33. P. Cordemans, J. Boydens, S. Van Landschoot, Embedded software development by means of remote prototyping, in *Proceedings of the 20th International Scientific and Applied Science Conference: Electronics-ET*, 2011
34. P. Cordemans, J. Boydens, S. Van Landschoot, E. Steegmans, Test-driven development strategies applied to embedded software, in *Proceedings of the Fifth European Conference on the Use of Modern Information and Communication Technologies*, 2012
35. M. Fletcher, W. Bereza, M. Karlesky, G. Williams, Evolving into embedded development, in *Proceedings of Agile 2007*, 2007
36. M. Karlesky, W. Bereza, C. Erickson. Effective test driven development for embedded software, in *Proceedings of IEEE 2006 Electro/Information Technology Conference*, 2006
37. M. Karlesky, W. Bereza, G. Williams, M. Fletcher, Mocking the embedded world: test-driven development, continuous integration, and design patterns, in *Proceedings of the Embedded Systems Conference 413*, 2007
38. S. Burbeck, Applications programming in smalltalk-80(tm): how to use model-view-controller (mvc) (1992), http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

A Fuzzy Cuckoo-Search Driven Methodology for Design Space Exploration of Distributed Multiprocessor Embedded Systems

Shampa Chakraverty and Anil Kumar

Abstract This chapter presents a methodology for conducting a Design Space Exploration (DSE) for Distributed Multi-Processor Embedded systems (DMPE). We introduce the notion of a *Q-node* to include quality-scaled tasks in the application model. A fuzzy rule-based requirements elicitation framework allows the user to visualize and express the availability requirements in a flexible manner. We employ *Cuckoo Search* (CS), a metaheuristic that mimics the cuckoo birds' breeding behavior, to explore the multi-objective design space. A fuzzy engine blends together multiple system objectives *viz. Performance, Qualitative Availability* and *Cost-Effectiveness* to calculate the overall fitness function. Experimental results illustrate the efficacy of the DSE tool in yielding high quality architectures in shorter run times and with lesser parameter tuning as compared with genetic algorithm. The fuzzy rules approach for fitness evaluation yields solutions with 24 % higher availability and 14 % higher performance as compared with a conventional approach using prefixed weights.

1 Introduction

The *Electronic Design Automation* (EDA) industry has ushered in the era of pervasive computing where digital devices are indispensible for executing every aspect of modern civilization. Several EDA tools such as Vista from Mentor Graphics [1], EDA360 from Cadence [2], Platform Architect from Synopsys [3] and Simulink from Mathworks [4] are available for carrying out high level as well as low level system

S. Chakraverty (✉)
Netaji Subhas Institute of Technology, Dwarka, Sector 3, New Delhi 110078, India
e-mail: shampa@ieee.org

A. Kumar
Samsung Research Institute, Noida, India
e-mail: anilk@ieee.org

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_5,

design. The field of EDA has been richly researched upon and utilized extensively by both academia and industry. Nevertheless, there are certain critical issues that specifically relate to the design of multi-objective DMPE systems. These issues need to be addressed more systematically and in a manner that supports active user participation in quantifying tradeoffs between conflicting design objectives.

Multi-processor design optimization falls under the category of NP-Complete problems. Not only does it confront a very large search space but it has also to deal with several design objectives simultaneously. The system must be designed so as to achieve the desired real time performance levels, ensure a high degree of availability and accuracy at its service points and possess the ability to reconfigure in the presence of faults. All these deliverables must be achieved in a cost effective manner. Another level of complexity arises from the sheer diversity of implementation platforms that are available for executing the functional tasks. They include software implementations on a range of *Instruction Set Architecture* (ISA) based processors, hardware implementations on *Application Specific Integrated Circuits* (ASIC) and bit map implementations on *Field Programmable Gate Arrays* (FPGA). These are matched by the plethora of bus types that are available to build the underlying communication fabric with.

Literature is rife with examples of traditional mathematical and graph based approaches for system optimization with focus on reliability. They include *Mixed Integer Linear Programming* (MILP) [5], dynamic programming [6] and [7], integer programming [8] and branch-and-bound [9] techniques. These approaches take a very long run time to generate optimal architectures for even medium sized applications. Multi-objective design optimization problems have been tackled practically by utilizing metaheuristic optimization algorithms. Meta-heuristic techniques do not guarantee the best solution but are able to deliver a set of near optimal solutions within a reasonable time frame. They generate an initial set of feasible solutions and progressively move towards better solutions by modifying and updating the existing solutions in a guided manner. Several metaheuristics such as *Simulated Annealing* (SA) [10, 11], *Tabu Search* (TS) [12], *Ant Colony Optimization* (ACO) [13, 14], *Practical Swarm Optimization* (PSO) [15, 16] and *Genetic Algorithm* (GA) [17–19] have been used for reliable multiprocessor design optimization. However, these approaches also suffer their own drawbacks. For example SA suffers from time longevity for a low temperature cooling process. GA is an effective optimization technique for large sized problems and can efficiently deal with the problem of convergence on local sub-optimal solutions. But GA requires excessive parameter tuning. In general existing approaches face difficulty in fitting into design exploration semantics because of their random and unpredictable behavior. We have used Cuckoo Search (CS), a new optimization algorithm developed by Yang and Deb that is by inspired by the unique breeding behavior of certain cuckoo species [20], for our multi-objective DMPE design optimization. Experiments demonstrate that it is able to deliver a set of high quality optimal solutions within a reasonable period. Besides, it needs minimal parameter tuning.

An important issue that needs due consideration is—*how to harness the end-user's proactive participation in the process of system design?* Reliability and Availability

(R&A) are particularly customer-oriented quality parameters. Therefore we need to develop a user-centric design methodology. Design objectives are often characterized by a degree of uncertainty and imprecision. Sophisticated applications with fault tolerant capabilities embody a fair degree of complexity, making it even more difficult to quantify in a precise manner their multiple qualitative requirements and to formulate an objective function that encapsulates all the requirements and their interdependencies in an exact and deterministic way. We adopt a two-pronged strategy to tackle this situation. Firstly, a fuzzy rule-based requirements elicitation system allows the user to describe her myriad availability requirements in a flexible and realistic manner. Secondly, a fuzzy model of the optimization function smoothly blends together the multiple objectives to evaluate the overall fitness function of solutions. The quality of solutions obtained can thus be validated against the requirements captured.

In this chapter we present a Fuzzy Cuckoo Search driven Design Space Exploration (FC-DSE) for multi-processor systems that tackles the issues we have broached upon. Section 2 describes the design environment for FCS-DSE. Section 3 elucidates the FCS-DSE methodology. Section 4 presents salient experimental results. We conclude the chapter in Sect. 5.

2 Design Environment

The proposed FCS-DSE system is a platform driven co-synthesis framework. It entails a synergetic hardware software design approach with a user-friendly interface. The design environment considers various functional and qualitative requirements of a given application. It refers to a library of feasible technological options for realizing the processing and communication needs of the computing system. Using these inputs, the co-synthesis system explores through the design space of heterogeneous multiprocessor systems knit together by shared communication busses. Finally, it culls out the best architectural solutions that satisfy the specified qualitative requirements of the application. The core challenges involved in the DSE process are:

- *Resource selections*: Since the hardware software system must be constructed from scratch, the choice of its basic building blocks *viz.* its processing units and communication links must be of good quality. The starting point is to allocate a pool of these resources to pick and choose from.
- *Task assignment*: The next step is to distribute the tasks of an application among the selected resources in a judicious manner. This step is crucial because a proper mapping maximizes the cost savings achieved by resource sharing while ensuring that all quality parameters are being satisfied.
- *Task scheduling*: All tasks must be scheduled to satisfy their stipulated partial orderings, output deadlines as well as the sequential execution constraints due to shared resources.
- *Architecture optimization*: The optimization algorithm must generate a set of architectural solutions that satisfy the specified objectives.

The FCS-DSE system starts with the following inputs:

2.1 Application Model

The application is modeled as a *Conditional Task Precedence Graph* (CTPG) = $G(V, E)$. In this chapter, we have used a 12-node CTPG, TG_{12} shown in Fig. 1 as a running example to explain the DSE concepts and demonstrate experimental results. The set of nodes V represent the application's computation tasks. The set of edges $E \subseteq \{(v_i, v_j) \mid (v_i, v_j) \in V\}$ represent the communication tasks that carry data from a source task v_i, to the sink task v_j. External inputs are applied at the primary inputs. The application's services that are utilized by the end user are presented at the primary outputs. Their timely generation, accuracy and availability are of prime concern to the user.

Node Type: There are three kinds of nodes in the CTPG. In Fig. 1, the input paths of *AND* nodes are depicted as braced by a single arc. These tasks execute only when all input paths are present. The *OR* nodes, shown without any brace, execute when at least one of the inputs is present. We introduce a new type of quality-scaled Q node, shown in Fig. 1 with their input paths doubly braced. The input paths of a Q node contribute different degrees of accuracy to the sink task.

Weights are annotated at various points on the CTPG. There are two kinds of edge weights. One set of edge weights represent the data volumes carried between inter-communicating tasks. Another set of edge weights denote the accuracy levels contributed by the inputs paths of a node. For the sake of clarity in Fig. 1 we omit the data volumes and show only the accuracy levels on the edges of TG_{12}. The input arrival times are labeled at the primary inputs. The deadline constraints of the services are labeled on the primary outputs.

Timing Parameters: Our framework uses a stochastic timing model. The task execution times, the volumes of data transferred between two interacting tasks and thereby the data communication times and the input data arrival times are all considered to be bounded random variables. Only the output deadlines are fixed. Each of these timing parameters is modeled as a *Beta distribution* with four defining parameters: lower limit *ll*, upper limit *hl* and the shape parameters α and β. The execution time distributions are obtained beforehand by extensive task profiling. First, we obtain a frequency distribution of each task's execution time on a given processor by randomly varying its input data as widely as possible. Next, a curve fitting technique such as the Chi-squared estimation is used to extract the parameters of the Beta distribution [21].

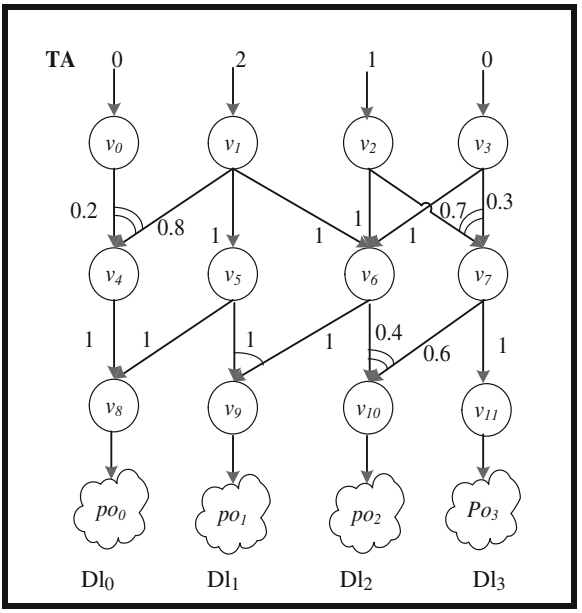


Fig. 1 A 12 task precedence graph

2.2 Technology Library

The technology library is a repository of technical data pertaining to the range of implementation platforms available. This includes databases for different types of *Processing Element* (PEs), the task execution times on each platform and different types of *Busses* (B).

(a) *Processor database*: The Processor database contains technical specifications of the PE types. Table 1 illustrates a representative processor database for the PE types used in our experiments. A task can exist as a software code running on an *Instruction Set Processor* (ISP). The ISP can be either a *General Purpose Processor* (GPP) catering to a wide range of tasks such as an IA-64 processor or it can be a *Special Purpose Processor* (SPP) that executes a limited range of domain specific tasks such as a Network Processor. The task can also be executed on a hardware platform such as an ASIC. Another alternative is to generate a bit-map configuration for the task and program it on an FPGA. The data for each processor type includes its failure rate, repair rate, the set of external busses that are supported by the PE type and its cost. The failure rates shown in Table 1 reveal that the reliability and cost of ASIC is highest followed by SPP and then GPP.

Table 1 Database for processing element types

PE type	Cost	Busses supported	Failure rate	Repair rate
GPP g_0	10	$b_0b_1b_2$	0.0005	0.5
SPP s_0	30	$b_0b_1b_2$	0.00009	0.06
ASIC a_0	50	b_0b_1	0.00002	0.09
ASIC a_1	70	b_0	0.00001	0.013

Table 2 Database for bus types

Bus type	Cost	Failure rate	Repair rate	Speed
b_0	5	0.0006	0.09	0.5
b_1	10	0.0002	0.05	1
b_2	12	0.0006	0.5	2

Table 3 Task execution times for TG_{12}: Lower Limit and Higher Limit. The α and β parameters (not shown) are assumed to be equal to 0.5 each

	GPP g_0	SPP s_0	ASIC a_0	ASIC a_1
v_0	5,9	7,8	*	*
v_1	5,7	4,4	3,3	*
v_2	*	7,8	3.7,4	5,5
v_3	*	7,7	6,6	4.5,5
v_4	8,10	7,8	*	3,3
v_5	11,15	9,12	7,7	*
v_6	*	9,10	7.3,8	3,3
v_7	10,15	8,12	7,7	*
v_8	4,5	3,4	*	*
v_9	9,15	8,10	6,7	*
v_{10}	11,16	*	7,8	*
v_{11}	14,17	14,15	9,9	6,6

(b) *Bus database*: The communication links of the distributed system are implemented using time shared buses. Table 2 shows the technical data for each bus type used by the DSE tool in our experiments. It includes its data transfer rate, failure rate, repair rate and its cost.

(c) *Execution Times database*: The probability distributions of the execution times for all tasks of the given application on each of the available PEs are stored in a database of execution times. Table 3 is the database of task execution times for the tasks of TG_{12} on the PE types in Table 1. Each entry in this table is a set of Beta distribution parameters for the corresponding execution time. The lower and upper limits of the timing distributions are shown in the table. The two shape parameters of each of these Beta distributions are assumed to be 0.5 each. A * denotes that it is infeasible to execute the task on the corresponding platform. Observe that the GPP takes the maximum time to execute a given task. The SPP takes lesser time while the ASICs incur the least execution time.

```
If
  (QL(po_n) is Low .AND. QL(po_n-1) is Medium .OR.....QL(po_1 )
   is Very High .AND. QL(po_0) is Remote)
then
  (IoA(po_n) is High, IoA (po_n-1) is Very High,..., IoA(po_0)
   is Remote).
```

Fig. 2 A sample fuzzy rule for the user's availability assessment

2.3 Availability Requirements Elicitation

The user ascribes different degrees of relative importance to the services available, called *Importance-of-Availability* (*IoA*). It is to be noted that these *IoA* values change under different situations. When the system is fully functional, the user assigns a perceived relative importance to each one of system's services. However, when one or more service(s) degrade in quality, the perceptions can change dramatically. The relative importance of the service points must then be assigned afresh.

The process of capturing the availability requirements begins when the user visualizes different scenarios depicting the quality level that is available at each primary output. The user's availability requirements are expressed in linguistic terms by a set of fuzzy rules. An example fuzzy rule is given in Fig. 2. It has an antecedent part and a consequent part. Its antecedent combines the various output quality levels $\{QL(po_i)\}$ using AND/OR operators. The set of output quality levels denotes a certain condition called *usage context*. The consequent part of the fuzzy rule assigns *IoA* weights to each of the primary outputs under the given usage context.

The user is free to input as many rules as deemed necessary to express all her availability requirements. These rules are stored in a database. Table 4 illustrates the set of fuzzy rules that were input for TG_{12} in our experiments. The topmost rule is valid for the fully functional scenario when each service point is available with its maximum accuracy (*QL is Very High*). The relative importance of the four outputs in this condition are such that po_0 and po_1 are considered less important that po_2 and po_3. Similarly, rules for other usage contexts are stored in the database.

Complimentary to the fuzzy framework for *IoA* requirements elicitation, our co-synthesis scheme incorporates a fuzzy engine to compose the multiple design objectives during the optimization process. We shall describe its working in the next section.

Given the above inputs *viz*. the RT application to be realized in the form of a CTPG, the platform library comprising the needed technological databases and the availability requirements described by a set of fuzzy rules, the FCS-DSE system launches a design space exploration process that evaluates candidate architectural solutions. The process finally yields a set of cost-optimal architectures.

Table 4 Fuzzy rules for user input contextual importance-of-availability requirements for TG_{12}

Rule no	*QL(po)* AND-ed combination				IoA(po)			
	po_0	po_1	po_2	po_3	po_0	po_1	po_2	po_3
1	*VH*	*VH*	*VH*	*VH*	*H*	*H*	*VH*	*VH*
2	*H*	*H*	*L*	*L*	*H*	*H*	*L*	*L*
3	*R*	*M*	*H*	*R*	*H*	*H*	*H*	*R*
4	*M*	*M*	*M*	*M*	*H*	*H*	*M*	*L*
5	*L*	*L*	*L*	*L*	*H*	*M*	*L*	*R*
6	*VH*	*H*	*H*	*VH*	*L*	*H*	*VH*	*VH*
7	*H*	*M*	*L*	*R*	*H*	*H*	*M*	*R*
8	*L*	*L*	*L*	*L*	*H*	*M*	*M*	*R*
9	*M*	*M*	*M*	*H*	*H*	*M*	*H*	*H*

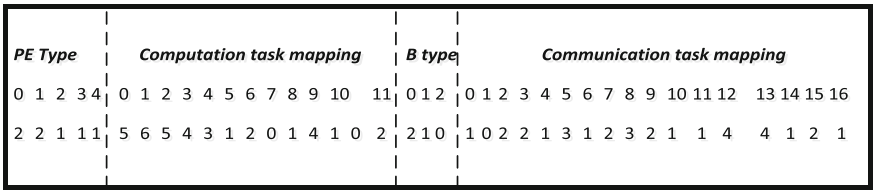


Fig. 3 A representative solution for CTPG TG_{12}

2.4 Architecture Representation

An architectural solution for the given CTPG and associated databases is encoded as a vector of integer values that define its architectural features. Figure 3 represents a feasible solution encoding for TG_{12}.

The solution encodes the following features:

1. The number of instances of each PE type: $\{N_{PE}\}$
2. The computation task (or node) to resource mappings $\{N_{V-P}\}$
3. The number of instances of each Bus types $\{N_B\}$
4. The communication task (or edge) to bus mappings $\{N_{E-B}\}$.

Thus the total number of decision variables L is equal to:

$$L = |N_{PE}| + |N_{V-P}| + |N_B| + |N_{E-B}| \tag{1}$$

3 Design Space Exploration

The FCS-DSE system uses the *Cuckoo Search* (CS) algorithm to optimize the distributed multiprocessor architectures. A set of global, qualitative objectives govern the optimization path. It embeds a fuzzy logic based fitness evaluator within CS to assess the fitness of the solutions.

In this section, we will first elucidate the various design objectives. Next we will elaborate upon the fuzzy model that is employed for composing the multiple design objectives. Finally we will explain the adaptation of the Fuzzy-CS metaheuristic to our DMPE design problem.

3.1 Design Objectives

1. *Real Time Performance*: We employ a non-preemptive scheduling algorithm that uses identities from probability theory to handle stochastic timings to assign start and completion times to each task. It uses an ALAP-ASAP list scheduling algorithm and a priority scheme that is based on the *timeliness* of tasks to meet prescribed deadline constraints.
   The system transits from a fully functional state to a partly functional state whenever any of its resources fail and undergoes a reverse transition when the failed resource is repaired by replacement. The performance of the system varies for each state. Let the system be in any given functional state S_k. Performance is calculated in terms of *Deadline Meeting Ratio* (*DMR*), defined as the ratio of the number of services d_k that are able to meet their deadlines to the total number of services N_{po}. A firm deadline constraint is imposed by incorporating the condition that if d_k is less than a pre-fixed number m, then *DMR* is taken to be zero. Thus the performance of the architecture in any given state S_k is given as:

$$Perf_k = \left[d_k < m?0 : \frac{d_k}{N_{po}} \right] \tag{2}$$

2. *Qualitative Availability*: We define a metric *Qualitative Availability* (QA) that integrates the notions of *Availability* and *Accuracy* (i.e. output quality). A *Continuous Time Markov Chain* (CTMC) model captures the underlying fail-repair process of the multiprocessor system. In essence, the system transits between different states of functionality. The steady-state probability P_k of being in each state S_k is calculated with the help of the CTMC model. In the fully functional state, all outputs deliver their maximum accuracy levels. When the system enters a partly functional state due to a fault, some of system's services may suffer degradation in their quality levels. We urge the reader to refer to [22] for a detailed description.
   As explained in Sect. 2, the user hypothesizes various combinations of service quality levels and accordingly prescribes fuzzy rules to re-assign changed importance values (*IoA*) to those services. In the face of a failure, a subset of these rules becomes applicable. A fuzzy engine infers and combines these fuzzy rules to generate a final, crisp *IoA* value to each primary output. The overall importance IoA_k of any partly functional state S_k is the summation of each of its output's $IoA_k(o_x)$ value normalized by the summation of the fully functional state's output importance values, $IoA_{FF}(o_x)$.

$$IoA = \frac{\sum_{x} IoA_k\,(o_x)}{\sum_{x} IoA_{FF}\,(o_x)} \tag{3}$$

The overall system Qualitative Availability QA_{sys} is given by the following equation.

$$QA_{sys} = \sum_{k} IoA_k * P_k \tag{4}$$

The aim of DSE is to maximize QA_{sys} and thereby ensure that the system continues to serve user perceived critical services even in the presence of faults.

3. *Cost_Effectiveness*: The cost of realization *Ct* of a chromosome is the cumulative cost of each resource deployed for realizing the architecture. The cost factor includes a feasibility constraint whereby the budget $Ct_{\max}$ should not be exceeded. In addition there is a cost minimization objective. The *Cost-effectiveness* of a solution is defined by the probability that the system is realizable under the prescribed budget $Ct_{\max}$. It is given by:

$$Cost\_Effectiveness = \left(\frac{Ct_{\max} - Ct}{Ct_{\max}}\right) \times U(Ct_{\max} - Ct) \tag{5}$$

where $U(.)$ is the unit step function.

3.2 Fuzzy Fitness Evaluator

Multi-objective design optimization has traditionally been dealt with by calculating the *Objective Function* (OF) or fitness of a solution either by using fixed weights to combine different objectives or by using the concept of Pareto-optimality to promote all non-inferior solutions. However, the level of sophistication in modern applications has surpassed the stage where one can formulate an OF that encapsulates all the requirements and their interdependencies in an exact and deterministic way. In our DSE scheme, we use a fuzzy model that allows the user/designer to prescribe a set of fuzzy rules that seamlessly blend together multiple objectives. They rules are processed by the Fuzzy Fitness Evaluator (FFE) engine.

Fuzzy Rules: Overlapping Fuzzy Sets {*Remote, Low, Medium, High, Very High*} are defined on each of the *Design Objectives* (input variables) as well as on the overall *Fitness* of the solutions (output variable) [26]. A set of fuzzy rules relate the composite condition formed by the AND/OR combination of objectives in their antecedent to the impact of this condition on the solution fitness in their consequent. Figure 4 illustrates a set of rules that combine three design objectives: the system's overall *Performance*, its *Qualitative Availability* (*QA*) and *Cost_Effectiveness* to determine the overall solution *Fitness*.

```
Rule 1:
if (Performance  is Very High .OR. Cost_Effectiveness is
Very High .OR. Qualitative Availability is Very High)
then (Fitness is Very High)
Rule 2:
if (Performance  is High .AND. Cost_Effectiveness is Me-
dium .AND. Qualitative Availability is Very High)
then (Fitness is Very High)
Rule 3:
if ( Performance  is Medium .OR. Qualitative Availabil-
ity is High .AND. Cost_Effectiveness is High)
then (Fitness is High)
```

Fig. 4 Examples of fuzzy rules for evaluating solution fitness by blending multiple objectives

Table 5 Fuzzy rules for fitness evaluation

Qualitative availability	Performance	Cost effectiveness	Overall fitness
AND-ed combination			
H	*H*	*H*	*H*
VH	*H*	*H*	*VH*
M	*VH*	*H*	*H*
H	*H*	*L*	*M*
L	*L*	*H*	*R*
R	*H*	*H*	*R*
M	*M*	*H*	*M*

Legend VH Very High; *H* High; *M* Medium; *L* Low; *R* Remote

Notably, rules can be framed to depict a set of criteria based on the concept of non-dominance as exemplified by Rule 1. The antecedent of this rule encapsulates a condition where at least one the objectives is *Very High*. In its consequent, it implies *Very High* solution fitness under this condition. In contrast, Rule 2 prescribes a linguistically weighted combination of various objectives. Rule 3 describes a composite condition by using both AND and OR operators. The user can specify many such rules in a flexible and discernible manner to guide the evaluation of solution fitness. Table 5 shows the set of rules that were input to the DSE tool for composing the design objectives and evaluating the fitness of solutions for TG_{12}.

Membership Function: In [23], Kasabov defined standard types of membership functions such as *Z* function, *S* function, trapezoidal function, triangular function and singleton function. These membership functions define the degree to which an input or output variable belongs to different but overlapping fuzzy sets. We chose the trapezoidal membership function as it is sufficient to capture the imprecise relationships between various objectives and is computationally simple. Figure 5 illustrates

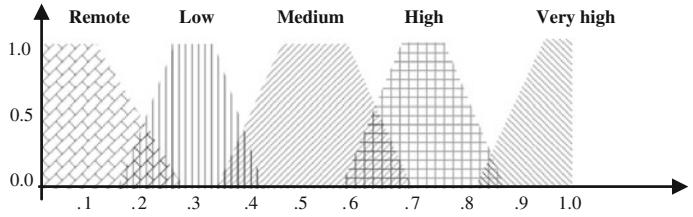


Fig. 5 Trapezoid membership function for the input and output variables of the fuzzy fitness evaluator

the trapezoid membership function that have been used to fuzzyfy all input objectives and the solution fitness.

Working of the fuzzy engines: We adopt the Mamdani model for the fuzzy engine employed to capture the *IoA* requirements of the application and also for the FFE sub-system.

1. *Fuzzification*: The absolute values of input variables are mapped into the fuzzy sets. The degree of membership of each input variable in each of the fuzzy sets is determined by consulting its membership function.
2. *Rules Activation*: For a given set of crisp values of design objectives, several rules may be invoked because each crisp value maps to more than one linguistic set. The overall strength of a rule is determined by the value of the membership functions of all the design objectives represented in its condition. The *min()* function for conjunctive *AND* operators and the *max* function for the disjunctive *OR* operator between conditions is used to determine input matching degree of a rule.
3. *Inferring the consequent*: The *clipping method* is used to infer the rule's conclusion. For each component in the consequent, its membership above the rule's matching degree is cut off and maintained at a constant level.
4. *De-fuzzification*: De-fuzzification converts the fuzzy fitness values to a crisp overall fitness. The *Centroid Of Area* method is used to generate a final crisp value for the output variables.

3.3 Cuckoo Search Driven DSE

Cuckoo Search (CS) is a metaheuristic search technique proposed by Yang and Deb [20]. It evolves a population of solutions through successive generations, emulating the unique breeding pattern of certain species of the cuckoo bird. Further, it uses a heavy-tailed Levy flight probability distribution to generate its pattern of random walk through the search space [25]. The Levy flight is simulated with a heavy tailed Cauchy distribution. Levy flight driven random walk is found to be more efficient than that obtained by Uniform or Gaussian distributions. In nature, it is used to advantage by birds and animals for foraging.

```
1. FCS_DSE(CTPG, Pop_size, Max_Gens, P_a)
2. begin
3.   Generate an initial set of architectural solutions
     Eggs, for the given CTPG. |Eggs| = Pop_size.
4.   while (!(Convergence) AND (generation!=Max-Gens)){
5.      Evaluate solution fitness values using FFE
        engine.
6.      Sort solutions according to descending fitness.
7.      Choose a random solution Egg_c using Cauchy
          distribution
8.      Choose an architectural characteristic in Egg_c
        using Cachy distribution
9.      Generate a new solution by modifying the charac-
        teristic using Cauchy distribution .
10.     Choose another solution Egg_u using uniform dis-
        tribution
11.     if (Fitness(Egg_c) > Fitness(Egg_u))
          then replace egg_u with egg_c.
12.     With probability Pa replace the least fitness
          solution with a new solution built from
          scratch.
13.  } //while
14.  end FCS_DSE
```

Fig. 6 Pseudocode for the fuzzy Cuckoo Search driven design space exploration FCS_DSE function

The cuckoo lays its eggs in the nest of another bird that belongs to a different species. Over many generations of evolution, the cuckoo has acquired excellent mimicry to lay eggs that deceptively resemble the host bird's eggs. Closer the resemblance, greater are the chances of the planted eggs being hatched by the host. The cuckoo also times the laying and planting of her eggs cleverly so that they hatch earlier than those of the host bird. Being first to hatch, the fledgling destroys new born chicks of the host to further enhance its survival chances. However, once in a while the host bird discovers the cuckoo eggs and either destroys them or simply abandons the nest to build a new one.

The pseudo-code in Fig. 6 describes our FCS_DSE optimization algorithm.

- Any feasible architectural solution is an egg. The process starts with a set of *new* solutions—the cuckoo's eggs, laid one per nest (line 2). High quality solutions with greater fitness correspond to those cuckoo eggs that most closely resemble host eggs and therefore have greater chances of surviving. The subsequent steps are performed for each generation till either the best fitness stabilizes or the maximum number of generations is reached.

- The solutions are sorted according to their fitness values as calculated by the FFE engine (lines 4, 5).
- The solutions are advanced one step at a time by generating a new solution from an existing one (lines 6, 7, 8). The choice of a solution, its targeted architectural feature and the modification in its value are all done by local random walk whose steps are drawn from heavy tailed (Levy flight) Cauchy distributions. Levy flight predominantly creates new solutions in the vicinity of good ones (exploitation), but resorts to sudden bursts of variations (exploration).
- If the new solution turns out to be superior to a randomly picked up existing solution, it replaces the old one (lines 9, 10, 11). This process is akin to a cuckoo chick (superior solution) destroying a host chick (inferior solution). It is an exploitative search technique which favors good solutions.
- With a probability *Pa*, a low quality solution with least fitness is discarded from the population and replaced another one built anew (line 12). This is analogous to the discovery of poorly mimicked cuckoo eggs by the host bird, abandoning the nest and building a new one from scratch. It helps the search process from getting stuck in local minima.
- Finally, high quality solutions are passed on to the next generation and the process is repeated. This ensures *survival of the fittest*.

4 Experimental Results

The Fuzzy Cuckoo Search driven Design Space Exploration (FCS-DSE) tool is implemented using Object oriented design in C++. We conducted our experiments on a Pentium dual core 2.59 GHz processor.

We input the synthetically generated 12-node CTPG TG_{12} shown in Fig. 1 to the FCS-DSE tool. TG_{12} is representative of real time applications that require a high degree of concurrency among its tasks and also impose some sequential constraints. The Processor and Bus databases are given in Tables 1 and 2 respectively. Table 3 shows the execution time distributions of the tasks of TG_{12} on available processor types, Table 4 gives the fuzzy rules for *IoA* requirements and Table 5 gives the fuzzy rules for fitness evaluation. The fitness evaluation rules in Table 5 give a higher preference to *Qualitative Availability* as compared to *Performance* and *Cost_Effectiveness.*

4.1 Task Allocation

Starting with a population of 20 chromosomes and the value of *Pa* set to 0.5, the design exploration was conducted through 400 generations. The task-to-processor mapping of the best architecture that was obtained at the end of exploration is shown in Fig. 7. The block diagram of the architecture is given in Fig. 8. The system uses

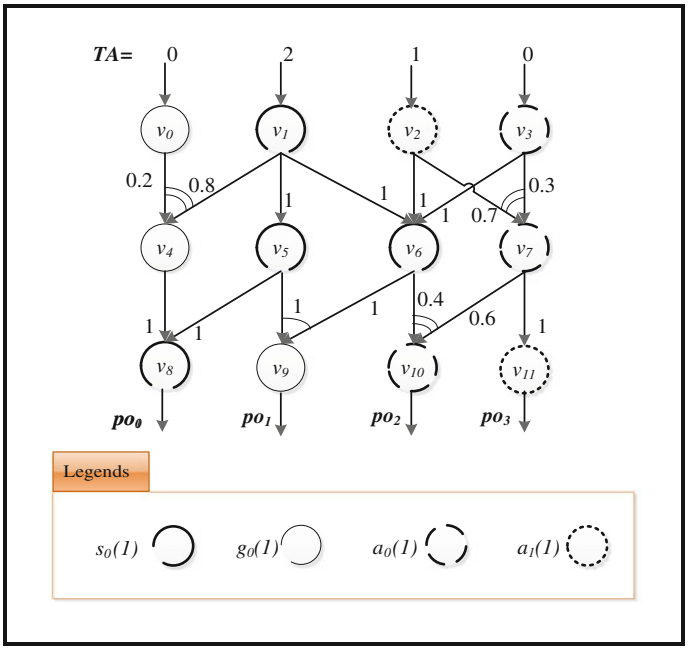


Fig. 7 Architecture of the best solution for TG_{12}

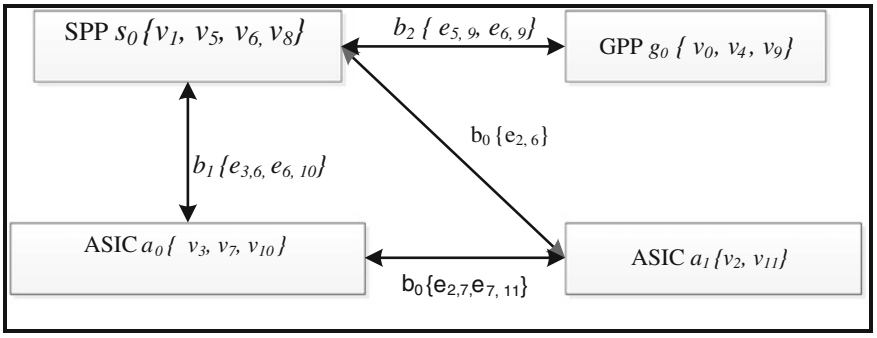


Fig. 8 Block diagram of the architecture

four processors to execute the computational tasks and four busses to implement the communication tasks.

The following allocation patterns can be observed:

- Sequential tasks and tasks at different hierarchical levels share the same processor. For example v_1 precedes v_5 which in turn precedes v_8. All three tasks share the same processor SPP s_0.

- Tasks at the same hierarchical level having equal precedence order are allocated to different processors, thus allowing their concurrent execution. For example the terminal tasks v_8, v_9, v_{10} and v_{11} are allocated to different PEs.
- Tasks that contribute to primary outputs with high *IoA* values are predominantly allocated to ASICs. The first rule in Table 4 shows that the primary outputs po_2 and po_3 are given more importance than others in the fully functional condition. Moreover they lose their importance drastically when their output quality level reduces (for example see rules 4, 5 and 8 in Table 4). Hence the system assigns the more reliable ASICs to implement the tasks v_2, v_3, v_7, v_{10} and v_{11} that contribute to po_2 and po_3.
- Tasks that contribute to less important outputs are mapped to less reliable, cheaper processors. For example tasks v_0, v_1, v_4, v_5, v_7 andv_9 contribute to outputs po_0 and po_1. Table 4 shows that these outputs have lesser important in the fully functional state (rule 1). Even when their *QL* levels diminish, these outputs are still acceptable (rules 4, 5 and 8). The DSE system aptly allocates a GPP for tasks v_0, v_4, v_9 and an SPP for tasks v_1, v_5, v_7 as they are less reliable and cheaper PEs than ASICs.
- For the best architecture, there are seven local inter-task communications and seven remote inter-task communications. The remote data transfers are implemented on four bus instances in a manner such that sequential edges share a bus ($e_{3,6}$ and $e_{6,10}$) while concurrent edges are assigned to different busses ($e_{2,6}$ and $e_{2,7}$).

The above observations indicate that the FCS-DSE system allocates the resources among the tasks of an application judiciously so as to enhance concurrency and availability in a cost-effective manner.

4.2 Route-to-Optimization: CS Versus GA

In this experiment, we compared the performance of the CS optimization algorithm with that of GA. GA is a widely used population based optimization algorithm that improves solutions by emulating the natural evolution of species through gene modification by crossover and mutation [24]. In these experiments, we set the probability of building solutions from scratch *viz Pa* as 0.25, 0.5 and 0.75 for three sets of experiments respectively. The experiments were then repeated using GA on the same data-set. For GA the *Mutation Rate* (*MR*) was set to 0.25 and 0.5 for two sets of experiments respectively and the crossover frequency was fixed at 0.4.

Average fitness improvement: The average fitness indicates the health of a population as a whole. As demonstrated in Fig. 8 and reported in [25], the average quality of population obtained through CS was better than that obtained through GA throughput the evolution process. Unlike GA that uses a uniform distribution for its random moves, CS mimics the Levy flight by utilizing the heavy tail characteristics of a *Cauchy* distribution. This allows a good balance of exploitation and exploration during evolution. There are relatively long periods of exploitation when random moves hover around the average of the Cauchy distribution. These are interspersed by bursts

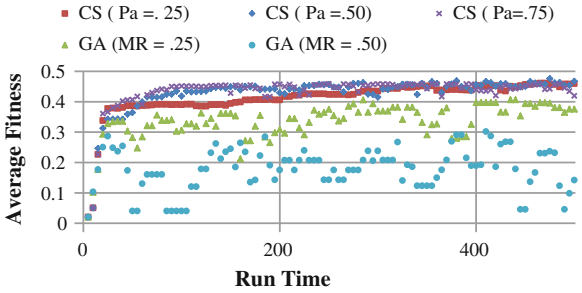


Fig. 9 Comparison of average fitness achieved by GA and CS

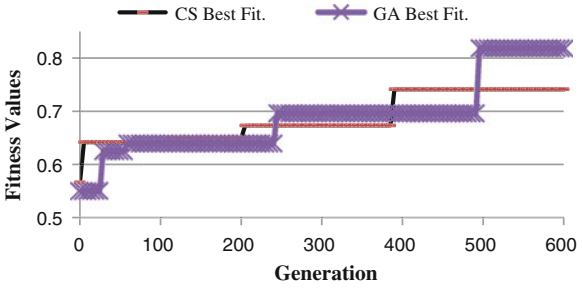


Fig. 10 Comparison of best fitness achieved by GA and CS

of exploitation when random moves are taken from the tail end of the distribution. Significantly, these transitions from exploitative search to explorative search occur naturally due to the characteristics of the heavy-tailed Cauchy distribution. In sharp contrast GA requires adjustment of two parameters *viz.* crossover rate and mutation rate to balance exploration and exploitation. CS thus relies on fewer tuning parameters (Fig. 9)

- *Best Fitness*: Figure 10 gives a plot of the best fitness value achieved in each generation for CS and GA driven optimization processes. The value of *Pa* for CS was set to 0.5 and *MR* for the GA was set to 0.25. The best fitness is a monotonically function across generations. This is due to the selection operator Elitism which ensures that the best solution in any generation is preserved in the next generation. We find that CS produces the better results in shorter run time. However, CS was not able to make any further change in the best solution during extended run time beyond 400 generations. In contrast to this, GA improved the fitness of best solution during the extended time period. This happened because CS looks for new solutions mostly in the vicinity of good solutions while GA explores the search space evenly with the help of random moves taken from a uniform distribution.

Table 6 Final solutions obtained through various fitness evaluation methods

	Probabilistic fitness function	Weighted sum fitness function	Fuzzy Logic based fitness function
Execution time	201 s	205 s	253 s
Availability	0.8143	0.7802	0.97078
Performance	0.75706	0.77345	0.88398
Cost feasibility	0.83096	0.5045	0.74
Overall fitness	0.512265	0.722608	0.81

4.3 Effectiveness of Fuzzy Fitness Evaluation

We experimented with the DSE tool to assess architectural solutions for TG12 by inserting three different fitness evaluation functions within the optimization code:

1. *Probabilistic fitness*: Objectives are expressed as probabilistic quantities and multiplied together.
2. *Weighted fitness*: Each objective is assigned a predetermined weight and summed up. The objectives $Perf_{sys}$ and QA_{sys} and CF_{sys} were given weights equal to 0.35, 0.45 and 0.2 respectively. Thus availability has the highest weightage.
3. *Fuzzy fitness*: Fuzzy rules are invoked to compose the objectives and calculate the overall fitness. The FFE sub-system that was described in Sect. 3 carries out this function. The rules given in Table 5 were used to evaluate the overall fitness. Note that these rules give greater preference to availability than performance or cost.

Table 6 shows the quality of best solutions obtained through the probabilistic, weighted and fuzzy based fitness evaluation methods for TG12. The fuzzy logic based fitness function has produced the best solution. Its system availability is 24.45 % higher than weighted fitness method and 19.24 % higher than the probabilistic fitness methods. Moreover its performance also surpasses that obtained by the weighted fitness method by 14.29 % and the probabilistic fitness method by 16.76 %. It is cheaper than the solution of the weighted fitness method by 46.67 % but costlier than the solution of the probabilistic fitness method by 10 %.

The above results highlight the fact that when fixed weights are used for each objective throughout the exploration path as in the case of weighted and probabilistic methods, then the optimization algorithm rejects several combinations of objectives that are actually acceptable to the user. Fuzzy rules evaluate the fitness values more faithfully, applying different criteria under different states of functionality. This gives a better chance to a wider variety of solutions to participate in the next generation of evolution.

5 Conclusion

In this chapter, we presented a user-centric design exploration methodology for designing multi-objective multiprocessor distributed embedded systems. The proposed methodology taps the power of a slew of soft computing techniques. The effective use of fuzzy logic provides users the flexibility to specify a range of availability requirements under the fully functional and various faulty situations. Fuzzy rules also allow the user/designer can express rules expressing acceptable trade-offs among various conflicting design objectives in terms of linguistic variables. These fuzzy rules are blended together smoothly by a fuzzy engine. We demonstrated the efficacy of a new optimization algorithm called Cuckoo Search for conducting the DSE. In comparison with GA, CS required lesser parameter tuning and was able to produce a range of higher quality solutions in shorter run times.

References

1. Mentor Graphics, Vista a complete TLM 2.0-based solution (2011), Available: http://www.mentor.com/esl/vista/overview. Accessed 2 June 2011
2. Cadence, A cadence vision: EDA360 (2011), Available: http://www.cadence.com/eda360/pages/default.aspx. Accessed 2 June 2011
3. Synopsys, Platform architect: SoC architecture performance analysis and optimization (2011), Available: http://www.synopsys.com/Systems/ArchitectureDesign/pages/PlatformArchitect.aspx. (Accessed 3 June 2011)
4. Simulink, mathworks. (Online)
5. R. Luus, Optimization of system reliability by a new nonlinear integerprogramming procedure. IEEE Trans. Reliab. **24**(1), 14–16 (1975)
6. D. Fyffe, W. Hines, N. Lee, System reliability allocation and a computational algorithm. IEEE Trans. Reliab. **17**(2), 64–69 (1968)
7. Y. Nakagawa, S. Miyazaki, Surrogate constraints algorithm for reliability optimization problems with two constraints. IEEE Trans. Reliab. **3**(2), 175–180 (1981)
8. K. Misra, U. Sharma, An efficient algorithm to solve integer programming problems arising in system-reliability design. IEEE Trans. Reliab. **40**(1), 81–91 (1991)
9. C. Sung, C.Y. Kwon, Branch-and-bound redundancy optimization for a series system with multiple-choice constraints. IEEE Trans. Relaib. **48**(2), 108–117 (1999)
10. B. Suman, Simulated annealing-based multi-objective algorithm and their application for system reliability. Eng. Optim. **35**(4), 391–416 (2003)
11. V. Ravi, B. Murty, P. Reddy, Nonequilibrium simulated annealing algorithm applied to reliability optimization of complex systems. IEEE Trans. Reliab. **46**(2), 233–239 (1997)
12. S. Kulturel-Konak, D. Coit, A.E. Smith, Efficiently solving the redundancy allocation problem using tabu search. IIE Trans. **35**(6), 515–526 (2003)
13. Y.-C. Liang, A. Smith, *An Ant System Approach to Redundancy Allocation*, in Proceedings of the 1999 Congress on Evolutionary Computation (Washington, D.C., 1999)
14. Y.-C. Liang, A. Smith, Ant colony paradigm for reliable systems design, in *Reliability Engineering*, vol. 53, ed. by Computational Intelligence (Springer, Berlin, 2007), pp. 417–423
15. G. Levitin, X. Hu, Y.-S. Dai, Particle swarm optimization in reliability engineering, in *Computational Intelligence in Reliability Engineering*, vol. 40, ed. by G. Levitin (Springer, Berlin, 2007), pp. 83–112

16. P. Yin, S. Yu, W.P.P. Wang, Y.T. Wang, Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization. J. Syst. Softw. **80**(5), 724–735 (2007)
17. P. Busacca, M. Marseguerra, E. Zio, Multiobjective optimization by genetic algorithms: application to safety systems. Reliab. Eng. Syst. Safety **72**(1), 59–74 (2001)
18. A. Konak, D.W. Coit, A.E. Smith, Engineering & system safety multi-objective genetic algorithms: a tutorial. Reliab. Eng. Syst. Safety, 992–1007 (2006)
19. L. Sahoo, A.K. Bhunia, P.K. Kapur, Genetic algorithm based multi-objective reliability optimization in interval environment. Comput. Ind. Eng. **62**(1), 152–160 (2012)
20. X. Yang, S. Deb, Engineering optimisation by cuckoo search. Int. J. Math. Model. Numer. Optimisation **1**(4), 330–343 (2010)
21. I. Olkin, L. Glesser, C. Derman, *Probability Models and Applications*, 2nd edn. (Prentice Hall College Div, NY, 1994)
22. K. Anil, C. Shampa, A fuzzy based design Exploration scheme for High Availability Heterogeneous Multiprocessor Systems. eMinds: Int. J. Human-Computer Interact **1**(4), 1–22 (2008)
23. N.K. Kasabov, *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering* (The MIT Press, Cambridge, 1996)
24. J. Anderson, *A Survey of Multiobjective Optimization in Engineering Design*, Technical Report Department of Mechanical Engineering (Linköping University, Sweden, 2000)
25. A. Kumar, S. Chakerverty, Design optimization for reliable embedded system using Cuckoo search, in IEEE International Conference on Electronics Computer Technology (ICECT), Kanyakumari, India, (2011)
26. L.A. Zadeh, Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)

Part III
Modeling Framework

Model-Based Verification and Validation of Safety-Critical Embedded Real-Time Systems: Formation and Tools

Arsalan H. Khan, Zeashan H. Khan and Zhang Weiguo

Abstract Verification, Validation and Testing (VV&T) is an imperative procedure for life cycle analysis of safety critical embedded real-time (ERT) systems. It covers software engineering to system engineering with VV&T procedures for every stage of system design e.g. static testing, functional testing, unit testing, fault injection testing, consistency techniques, Software-In-The-Loop (SIL) testing, evolutionary testing, Hardware-In-The-Loop (HIL) testing, black box testing, white box testing, integration testing, system testing, system integration testing, etc. This chapter discusses some of the approaches to demonstrate the importance of model-based VV&T in safety critical embedded real-time system development. An industrial case study is used to demonstrate the implementation feasibility of the VV&T methods.

1 Introduction

Real-time systems is one of the challenging research area today, which addresses both software and hardware issues related to computer science and engineering design. In a real-time system the correctness of the system performance depends not only on the logical results of the computations, but also on the time at which the results are produced [1]. A real-time system changes its state precisely at physical (real) time instant, e.g., maintaining the temperature of a chemical reaction chamber is a complex continuous time process which constantly changes its state even when

A. H. Khan (✉) · Z. Weiguo
Northwestern Polytechnical University, Xi'an, People's Republic of China
e-mail: arsalan1@mail.nwpu.edu.cn

Z. Weiguo
e-mail: zhangwg@nwpu.edu.cn

Z. H. Khan
Center for Emerging Sciences, Engineering and Technology (CESET), Islamabad, Pakistan
e-mail: zkhan@ceset.pk

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_6,

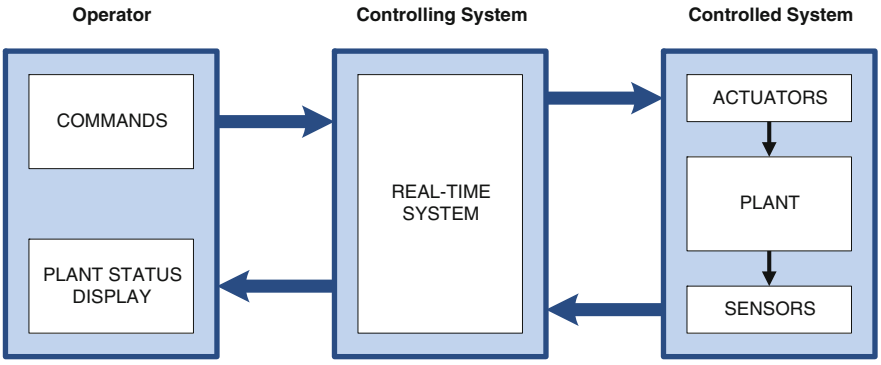


Fig. 1 Typical real-time system

the controlling computer has stopped. Conceived from controlling the real world phenomena, real-time systems are often comprised of the following three subsystems shown in Fig. 1.

Controlled system is the device (the plant or object), we want to control according to the desired characteristics. It also contains actuating devices i.e., motors, pumps, and valves, etc. and sensors i.e., pressure sensor, temperature sensor, navigation sensor, and position sensors, etc. Surrounding environmental effects (disturbances), sensors noise and actuators limits are also considered as a part of this subsystem.

Operator environment is the human operator, who commands the controlling system to control the output of the controlled system. It also contains the command input device i.e., keyboards, joysticks, and brake pedals, etc.

Controlling system is the real-time system or the controller which acquires the information about the plant by using sensors and controls it with actuators according to user demands under sensors imperfection and actuating device limitation considerations.

Real-time systems can be categorized based on two factors [2]. The factors outside the computer system classify the real-time systems as soft real-time, hard real-time, fail-safe real-time and fail-operational real-time systems. The factors inside the computer system classify the real-time systems as event-triggered, time-triggered, guaranteed-timeless, best-effort, resource adequate and resource inadequate real-time systems. Typically, in real-time systems, the nature of time is considered, because deadlines are instants in time. Safety-critical real-time systems are mainly concerned with the result deadlines based on the underlying physical phenomena of the system under control.

Locke [3] describes the classification of the real-time systems according to the cost of missing a desired deadline as shown in Fig. 2. In a *soft real-time system*, producing a result after its deadline will still be valuable to some extent even if the outcome is not as profitable as if the result had been produced within the deadline and it may be acceptable to occasionally miss a deadline. Examples of such systems include the flight reservation system, TV transmissions, automated teller machine (ATM),

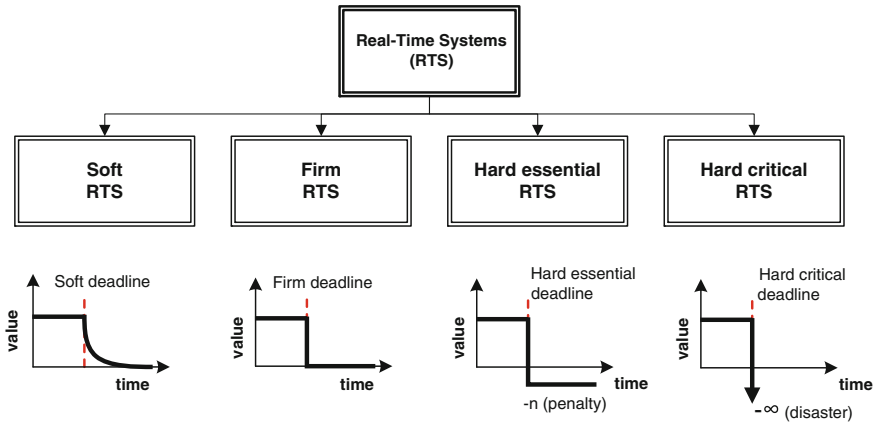


Fig. 2 Four types of real-time systems and their effects of missing a deadline

video conferencing, games, virtual reality (VR), and web browsing, etc. In a *firm real-time*system, producing a result after missing few deadlines will neither give any profit nor incur any cost but missing more than few may lead to complete or catastrophic system failure. Such systems include cell phones, satellite-based tracking, automobile ignition system and multimedia systems, etc. In a *hard essential real-time* system, a bounded cost will be the outcome of missing a deadline e.g., lost revenues in a billing system. Lastly, missing a deadline of a hard critical real-time system will have dreadful consequences e.g. loss of human lives and significant financial loss [4]. Examples of such systems include avionics weapon delivery system, rocket and satellite control, auto-pilot in aircraft, industrial automation and process control, medical instruments and air-bag controller in automotives to name few.

Hard critical real-time or simply *hard real-time* systems are usually safety-critical systems. These systems must respond in the order of milliseconds or less to avoid catastrophe. In contrast, soft real-time systems are not as fast and their time requirements are not very stringent. In practice, a hard real-time system must execute a set of parallel real-time tasks to ensure that all time-critical tasks achieve their specified deadlines. Determining the priority order of the tasks execution based on the provided rules is called *scheduling*. The time scheduling problem is also concerned with the optimal allocation of the resources to satisfy the timing constraints. Hard real-time scheduling can be either static (pre run-time) or dynamic in nature.

In *static scheduling*, the scheduling of the tasks is determined in advance. A run-time schedule is generated as soon as the execution of one task is finished by looking in a pre-calculated task-set parameters table, e.g., maximum execution times, precedence constraints, and deadlines. Clock-driven algorithms and offline scheduling techniques are mostly used in static systems, where all properties of the tasks are known at design time.

In *dynamic scheduling* there is no pre-calculated task-set, decisions are taken at run time based on set of rules for resolving conflict between tasks, we want to execute

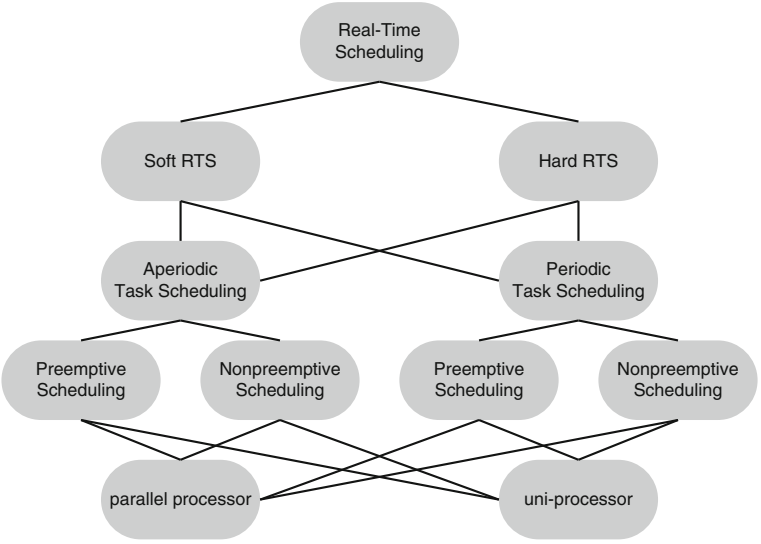


Fig. 3 Taxonomy of real-time scheduling with embedded hardware implementation

at the same time. One common approach is the introduction of pre-emptive and non-pre-emptive scheduling. Priority-driven algorithms are mostly employed in dynamic systems, with combination of periodic and aperiodic or sporadic tasks.

In *preemptive scheduling*, the presently running task will be preempted upon arrival of higher priority task. In *nonpreemptive scheduling*, the presently running task will not be preempted until completion. Parallel processing system is dynamic if the tasks can migrate between the processors and static if the tasks are bound to a single processor [5]. In general, the static systems have inferior performance as compared to dynamic systems in terms of overall job execution time. It is easy to verify, validate and test a static system as compared to dynamic system for which sometimes it may be impossible to validate the system. Because of this fact, hard real-time systems are preferred over static systems. Figure 3 shows the taxonomy of real-time systems scheduling with implementation architecture.

Often, the real-time systems are implemented with combination of both hard real-time tasks and soft real-time tasks. Traffic control system is a typical example of a critical hard real-time task to avoid crashes as compared to soft real-time tasks, where optimized traffic flows can be experienced. In measurement systems, value of timestamps is a hard real-time task and delivering timestamps is a soft real-time task. Another example of such an application is a quality control using robotics where defective product removal from the conveyer belt is a soft real-time task and stopping it in emergency is a hard real-time task.

Generally, embedded systems are used to meet the real-time (RT) system performance specifications in an individual processor form or in a complete subsystem form i.e., System-On-Chip (SOC). Dedicated embedded hardware i.e., gen-

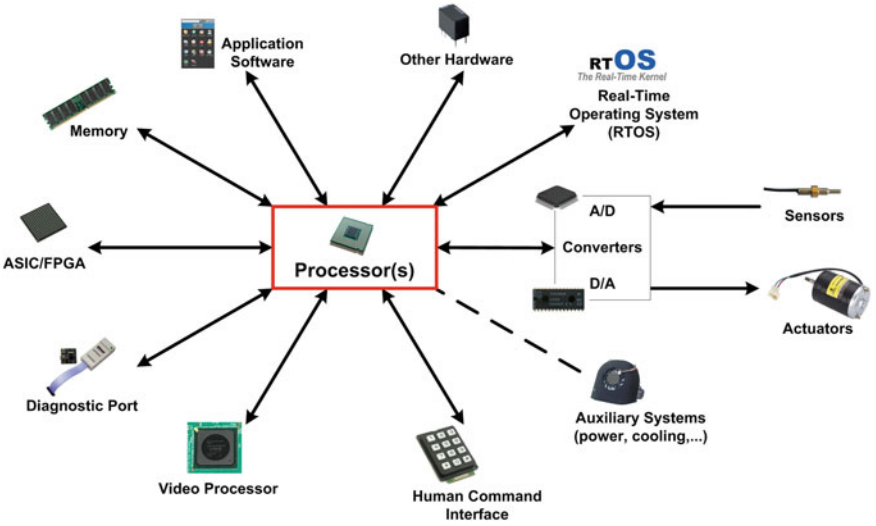


Fig. 4 Building blocks of embedded real-time system

eral microprocessors, micro-controllers, field programmable gate arrays (FPGAs), application specific integrated circuits (ASICs), and digital signal processors (DSPs), involved in sensing and control with real-time scheduling algorithms are used to deliver correct results at precise right time. Embedded real-time systems have advance computing capabilities, fault tolerance (FT), and quality-of-service (QoS), with additional constraints on size, weight, cost, resource optimization, time-to-market, power, and reliability in harsh conditions. According to recent studies, approximately 99 % of all the processors use in embedded systems. Main building blocks of embedded RT system are shown in Fig. 4.

Embedded systems are so common in our daily lives that even we do not notice their presence. Most common applications of embedded systems are shown in Fig. 5. Currently, multi-processor based embedded systems are widely used in communication systems, telecommunication and networking. According to recent research survey, software development for embedded systems has risen to account more than 50 % of the total project cost because of the development trend towards more and more use of multi-processor based architecture in next-generation embedded devices [6]. Multi-processor systems are quite fast as compared to single processor based systems but to ensure the reliability of these systems we still need more expertise in software development and verification, validation and testing (VV&T) of these systems.

Figure 6 shows the distribution of embedded systems in engineering applications. Communication systems have the largest share of embedded market because of the soft real-time requirements with multi-core processors and multiprocessors for high speed signal processing. According to a survey, software development engineers are expecting increase in multi-core processors usage in safety-critical applications

Fig. 5 Embedded real-time systems around us

such as those found in aerospace and medical equipment industries by employing advanced reliable verification, validation and testing procedures [6].

Since this chapter is mostly concerned with the safety-critical systems or hard real-time systems, it will use the term real-time system to mean embedded, hard real-time system, unless otherwise stated. Lack of expertise in embedded software development and insufficient advancement in VV&T methodologies/tools are main problems to cope, in order to transform the market trend towards multi-core processors architecture in safety-critical systems. VDC's 2010 Embedded System Engineering survey shows that current embedded projects using multiprocessors/multicore architectures have become larger, longer, and farther behind schedule than those utilizing single processor designs because of the unavailability of dependable model based VV&T tools. Figure 7 shows the importance of VV&T engineers in embedded real-time products development.

Software engineers have the major role in the development of modern technology. Due to the trend in next generation embedded real-time products and applications voracious consumer market large amount of embedded software is required to fulfill the customer demands. Embedded software quality, customer satisfaction, cost, and delivery on time are the four main factors to dominate in competitive market. As compared to late 1960s, now software engineering has improved a lot but still much advancement is needed in this area. As the software market expands and improves with the contribution of system engineering in ensuring the quality of product, the problems still exist because of the complexity of systems [7]. All the four objectives,

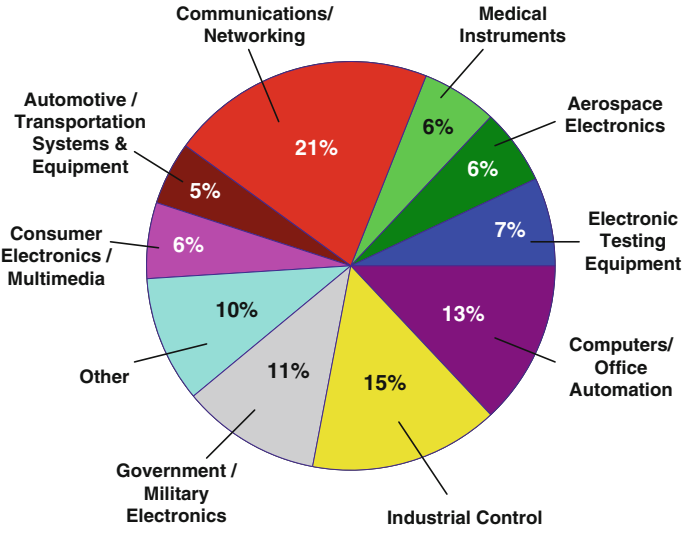


Fig. 6 Market shares of embedded systems in engineering systems

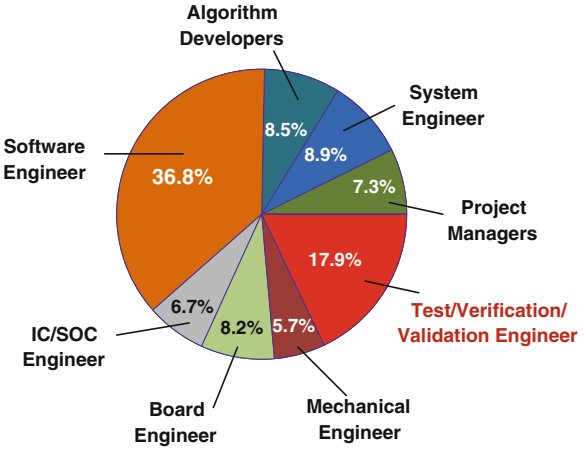


Fig. 7 Percentage of full-time engineers working on projects in the year 2010 (extracted from [6])

i.e. quality, user satisfaction, costs according to budget and schedule are interdependent as each corner of a square shown in Fig. 8. These interconnected targets can easily be achieved using model-based design and VV&T techniques, which is the theme of this text.

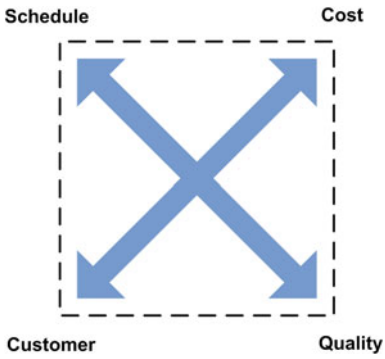


Fig. 8 Four objectives in next generation software engineering

2 Background and Purpose

One of the most demanding application of embedded real-time system is the safety-critical systems where little unseen software bug or hardware design malfunction can cause unenviable damage to the environment and could result in loss of life [8]. These expensive complex systems such as hybrid electric vehicles, autonomous underwater vehicles (AUV), chemical plants, nuclear power plants, aircraft, satellite launch vehicle (SLV) and medical imaging for non-invasive internal human inspections are designed in such a way to ensure system stability and integrity during all of the system functional modes in normal scenario and some level of performance and safe procedure in case of faults. Multiple distributed embedded real-time (RT) computers are used in medical, aerospace, automobile and industrial applications for fast real-time performance with controllability of the system.

Testing and qualification of a safety-critical embedded real-time system is of great importance for an industrial product development and quality assurance system. Embedded real-time system is a blend of advanced computer hardware and software to perform specific control function with stringent timing and resource constraints as a part of larger system often consists of other mechanical or electrical hardware. For example to maintain the flow of a liquid through a pipe, we need some actuating mechanism i.e., flow control valve and sensing device e.g., flow meter in a digital closed loop as shown in Fig. 9. Where r(t), r_k, y(t), y_k, e_k, u_k, and u(t) are reference command, sampled reference command, process output, sampled process output, sampled error, sampled control signal and continuous control signal respectively. At each phase of an embedded software development life-cycle, we require a quick verification methodology with appropriate validation test cases because of tight time schedules during product development.

The benchmark for measuring the maturity of an organization's software process is known as capability maturity model (CMM). In CMM there are five levels of process maturity based on certain key process areas (KPA) as shown in Fig. 10. There are four essential phases of embedded real-time software development process. First phase is of the requirements realization/generation, review, analysis, and specification.

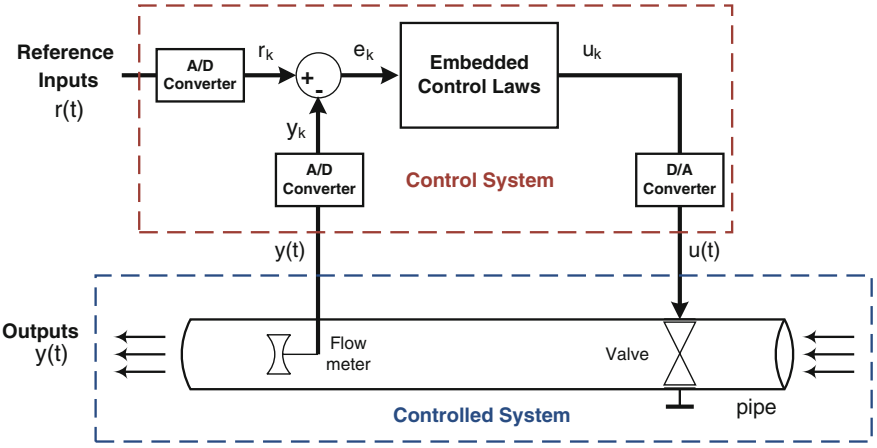


Fig. 9 Simplified block diagram of embedded real-time control system

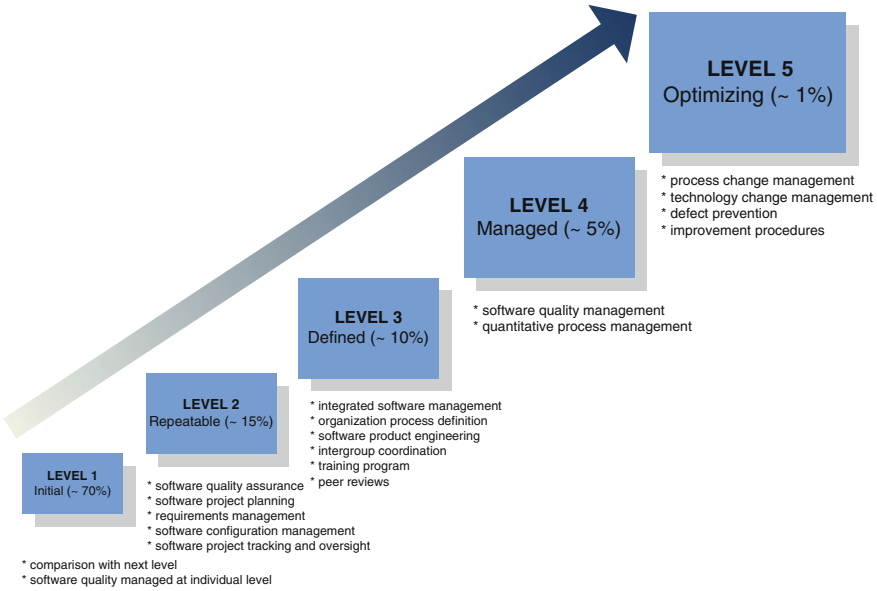


Fig. 10 Capability maturity model (CMM) for software development organizations

Second phase is of system design and review. Third phase includes algorithm implementation and final phase is of extensive testing. Each of these phases has an output which we have to validate. Interpretation of these phases may differ according to the projects. Each particular style and framework which describes the activities at each phase of embedded software development is called a "Software Development Life-Cycle Model".

3 Methods in VV&T

Here, we provide a brief overview of the model based verification, validation, and testing procedures employed for embedded system design. Verification is the process of assessing a system or component to determine whether the products of a given development phase satisfy the requirements imposed at the start of that phase whereas validation is the process of assessing a system or component during or at the end of the development process to determine whether it satisfies the specified product requirements [9]. Verification, validation and testing (VV&T) is the procedure used in parallel to system development for ensuring that an embedded real-time system meets requirements and specifications and that it fulfills its deliberate purpose. The principal objective is to determine faults, failures and malfunctions in a system and evaluation of whether the system is functional in an operational condition.

There are various embedded real-time software development life-cycle (SDLC) models available in technical literature. Some of the well known life-cycle models are:

1. Incremental Models
2. Iterative Models
   - Spiral Model
   - Evolutionary Prototype Model
3. Single-Version Models
   - Big-Bang Model
   - Waterfall Model without "back flow"
   - Waterfall Model with "back flow"
   - "V" Model

3.1 Incremental Models

In this software product development model, the product is designed, implemented, integrated and tested as sequence of incremental build as illustrated in Fig. 11. On the bases of initial requirements a partial implementation of complete system is performed. Then additional requirements and functionality is added. The incremental model prioritizes requirements of the system and then implements them in groups. Each subsequent incremental model of the system adds function to the previous increment as shown in Fig. 11.

3.2 Iterative Models

As compared to the incremental models, iterative models do not start with a complete initial specification of requirements and implementation. In fact, development

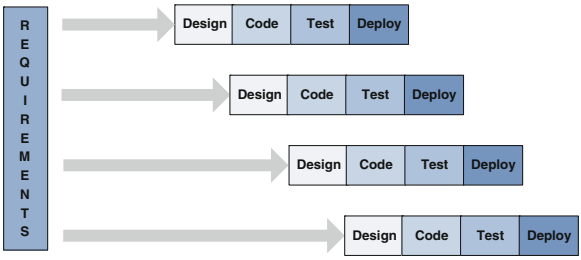


Fig. 11 Incremental SDLC model

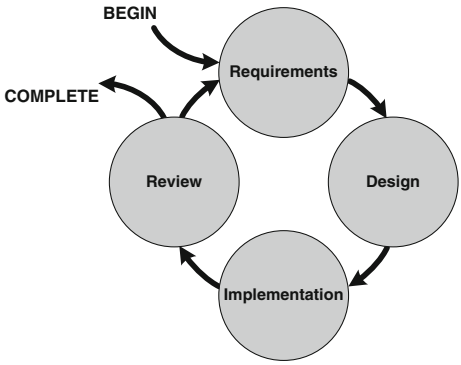


Fig. 12 Iterative SDLC model

begins with specifying and implementing just part of the software, which can then be reviewed and modified completely at each phase in order to identify further requirements. Figure 12 shows an iterative lifecycle model, which consist of repeating the four phases in sequence. For further details see Ref. [10].

3.3 Single-Version Models

In this software product models, one VV&T procedure is followed without addition or review of product design after requirements identification at later stage. Some of the models categorize in this group are discuss next.

3.3.1 Big-Bang Model

Here a software developer works in isolation for some extended time period to solve the problem statement. Developed product is then delivered to the customer with a hope that client is satisfied.

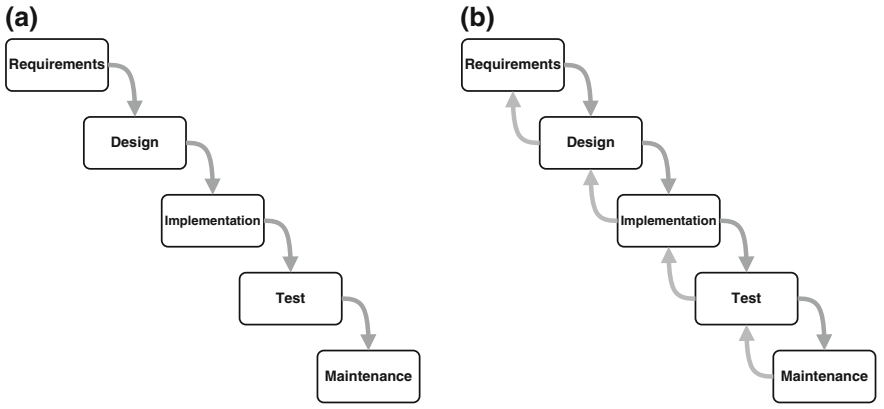


Fig. 13 Waterfall SDLC models. **a** Waterfall without "back flow", **b** Waterfall with "back flow"

3.3.2 Waterfall Models

This is one of the oldest and widely used classical software development models initially utilized in government projects. This model emphasizes on planning and intensive documentation which makes it to identify design flaws before development. The simplest waterfall lifecycle model consists of non-overlapping phases where each phase "pours over" into the next phase as shown in Fig. 13a.

Waterfall model starts with the requirements phase; where, the function, behavior, performance and interfaces are defined. Then, in the design phase; data structures, software architecture, and algorithm details are decided. In implementation phase the source code is developed in order to further proceed towards testing and maintenance phases. There are many variants of simple waterfall model. One of the most important is with correction functionality where detected defects are corrected by going back to the appropriate phase as shown Fig. 13b.

3.3.3 V-Model

One of the most effective and employed lifecycle model used for software design and VV&T is V-model [11] as shown in Fig. 14. In V-lifecycle model verification and validation activities are performed in parallel with development process. At initial development phase verification and validation are of static in nature whereas in later development phases they are dynamic [10]. The static verification is concerned with the requirements, analysis of the static system representation and tool based software analysis. The VV&T after availability of software is of dynamic character with test data generation, product functional performance testing, integrated system testing and acceptance testing.

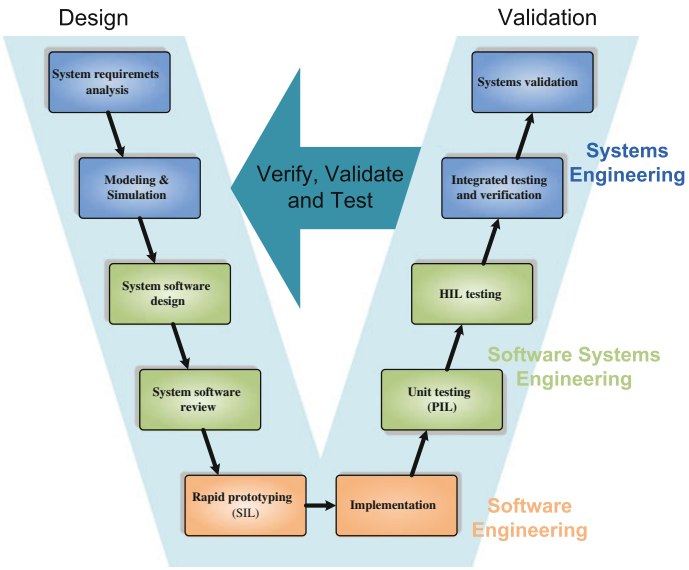


Fig. 14 Model-based embedded real-time software system development V-lifecycle model

The V-model can be considered as a unique product development model with VV&T process which considers the corresponding quality control activities, represented by big arrow from right to left part of the V in Fig. 14. The left part of V-model can be seen as the waterfall model from requirements description, requirements analysis, design definition, design review, down to implementation of software code on real embedded hardware, including subsequent verification activities. The right part of the V-model describes the testing activities, unit tests, HIL tests, integration tests, and system tests for qualification and acceptance. One of the important advantages of this product development process is that project management can track progress milestones. Each phases of this process model delivers a testified product. Another benefit of this model is that an inspection can be performed as soon as a product, or part of a product, is produced. Some typical examples are the inspection of developed algorithm against the design document and inspection of the part of a product through unit testing against requirements.

Here, we choose V-model to present for safety-critical embedded real-time system development because of its stability, reliability and easy to use characteristics. In literature, a lot of techniques and detail steps are available for verification, validation and testing in V-model which are related with software engineering, software systems engineering, and systems engineering, e.g.

1. Model checking
2. Inspection
3. Simulation
4. Static analysis

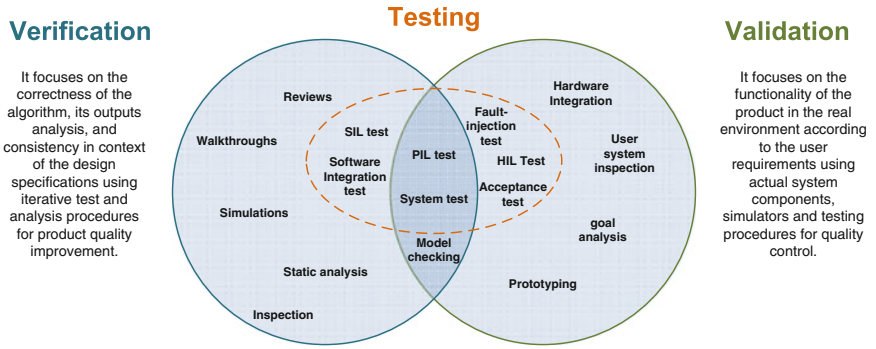


Fig. 15 VV&T techniques in V-model and their interconnection

5. Software-In-the-Loop testing (SIL)
6. White-box and Black-box testing
7. Processor-In-the-Loop testing (PIL)
8. Evolutionary testing
9. Hardware-In-the-Loop testing (HIL)
10. Fault-injection testing
11. Integrated and acceptance testing

and many more. For a quick survey of VV&T techniques and IEEE standard for software verification and validation see [12–14]. The interconnection between model based VV&T techniques in V-model is shown in self-explanatory Fig. 15.

Model checking: In model-based VV&T procedure, model checking is one of the most important techniques for analysis of complex safety-critical systems. Matlab/Simulink® [15] is one of the most widely used embedded real-time system model development, simulation, analysis, verification, validation, testing and rapid prototyping model-based tool chain for safety-critical applications. In Simulink® [16], stateflow charts and simulink models are used for system modeling as shown in Fig. 16. There are several toolboxes available in Simulink i.e., Simulink Design Verifier™, Simulink Verification and Validation™, SystemTest™, and Simulink Code Inspector™ for model checking and algorithm verification and validation according to DO-178D and IEC 61508 industry standards. Custom Simulink models can be developed using S-Function blocks.

Simulation: Simulation and model checking is performed repetitively against the defined requirements and specifications. It is also used for analysis purposes to check the design loopholes and test basis for final implemented software product.

Inspection: It is one of the verification techniques which are performed by group of people having experience in development and execution of the VV&T object (the product under study) and people having not. The team checks the VV&T object line by line against a checklist by means of its source code. Inspection includes walkthroughs, software reviews, technical reviews and formal inspection.

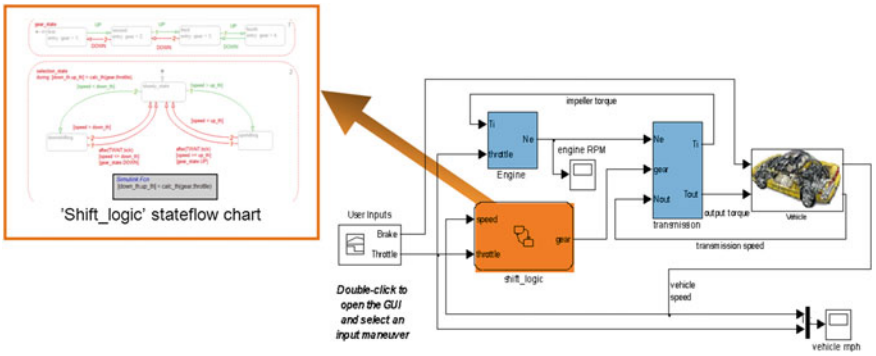


Fig. 16 Simulink automatic transmission control (ATC) system modeling

Static analysis: It means verifying a product without executing the VV&T object. These checks are text analysis, requirement analysis and VV&T functionality review. This review is performed by a team of VV&T experts and product designers. This technique is able to find missing, deficient and unwanted functionality in the source text of the product under analysis at early development phase.

Testing: Testing is carried out to check dynamically, whether the product requirements are fulfilled. Both, verification and validation can be performed by testing. The biggest advantage of testing as compared to other V&V techniques is the analyzing the system in realistic application environment. This allows the realistic online and real-time behavioral examination of the developed product. Here we describe some of the common In-the-loop testing procedures in model-based safety-critical embedded real-time product development.

Component testing: It is performed in SIL to verify the individual, elementary software component or collection of software components in a model-based closed-loop real-time simulation environment. Each software code is extensively verified with respect to requirements, timing and resource constraints. In model-based VV&T process automatic code generation technology is used for rapid prototyping of simulation model or design fully implemented in embedded software code. Which can also act as a final embedded real-time product or used as a simulator for further product testing procedures i.e., PIL, HIL and system integrated testing.

White box testing: It is called code based testing [17] which treats the software as a box that you can see into to verify the embedded real-time software code execution. *Black box testing* (also called specification based testing or functional testing [17]) treats software as a box that you cannot see into. Here we have to check the functionality of the implemented code according to the specification without consideration to how the software is implemented. It looks what comes out of the box when a particular input data is provided.

Processor-In-the-loop (PIL) testing: It is performed after successful implementation of design software on actual processing hardware with electronic hardware I/O interface availability. SIL simulation results are compared with the PIL simulation

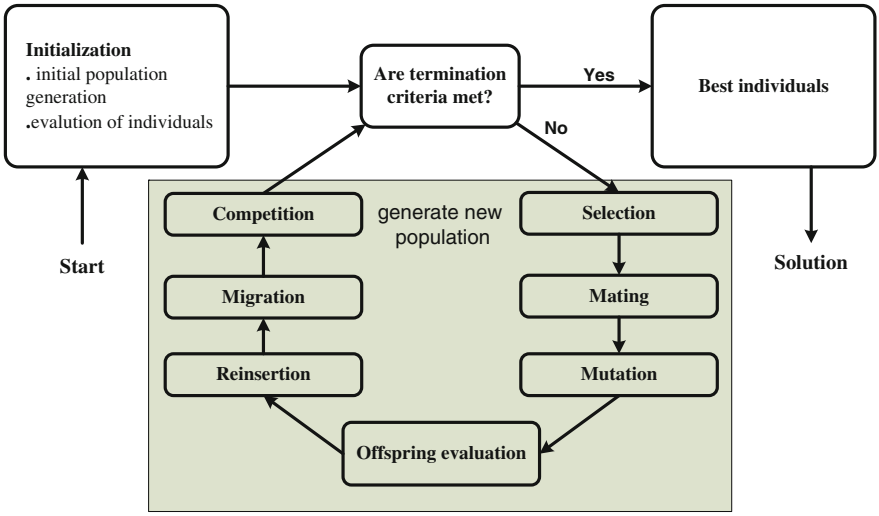


Fig. 17 Structure of evolutionary algorithm

results to verify the compiler and processor. In PIL, we check the shortest and longest execution times of the implemented embedded software algorithm through evolutionary testing. Where each individual of the population represents a test value for which the system under test is executed. For every test value, the execution time is measured to determine the fitness level of individual.

Extended evolutionary testing: In this approach is presented in [18] which allow the combination of multiple global and local searching strategies and automatic optimal distribution of resources for the success of the strategies. In recent study [19] evolutionary algorithm is used for generation of test traces that cover most or all of the desired behavior of a safety-critical real-time system. Evolutionary testing is also used for verification of developers' tests (DT). For a detailed discussion of evolutionary algorithms see [20]. General evolutionary algorithm execution flow is shown in Fig. 17.

Hardware-In-the-Loop testing: It is performed with realistic input data and other actual mechanical/electrical system components to validate the embedded real-time software. HIL testing is the standard verification and qualification procedure for safety critical products industry. HIL simulation response is compared with the PIL testing results and SIL testing results to ensure the correctness of the embedded real-time algorithm. Fault injection testing is also carried out with HIL simulation testing to check the robustness of the system to unwanted environmental conditions. Figure 18 illustrates the HILS testing of an unmanned aerial vehicle (UAV) to validate the flight controller.

After successful HIL testing and verification of embedded RT flight control computer (FCC), entire integrated system testing is performed. Where actual flight vehicle sensors, actuators and engine are installed with the FCC and the real-time flight

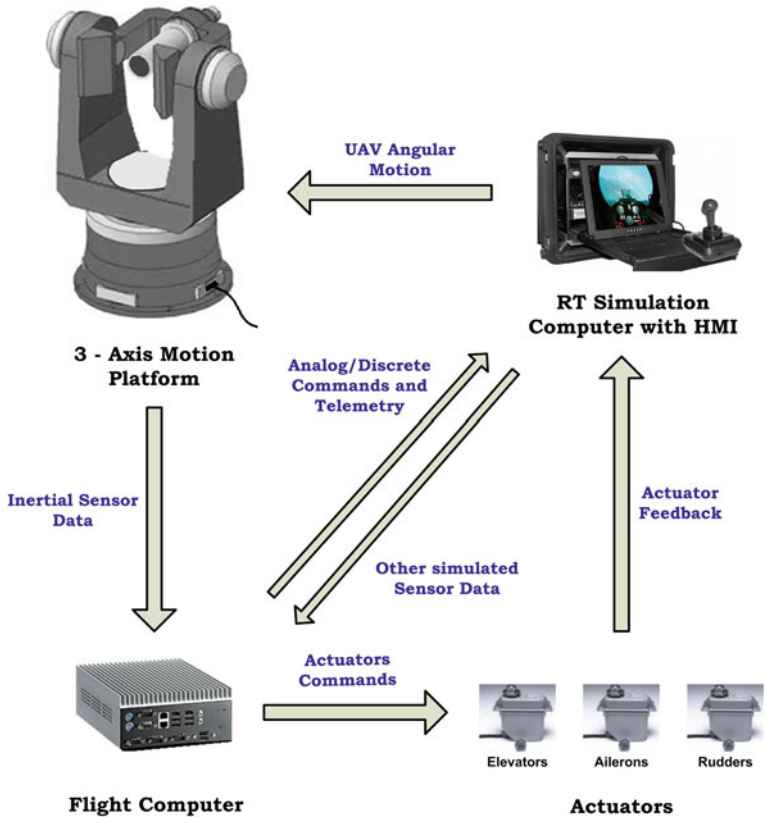


Fig. 18 HIL testing of embedded real-time flight controller

ground data is logged for verification and comparison with the design requirements and RT simulation. Integration testing checks the defects in the interfaces and interaction between integrated subsystems. These integrated subsystems behave as elements in an architectural design where the software and hardware works as a system [21].

Integrated and acceptance testing: Complete integrated system and formal acceptance testing is performed in the presence of customer to validate the system standard and customer requirements. After the certification and customer approval product will go to the production department with specification and VV&T details according to the requirement and quality standards.

4 Challenges in Model Based ERT Systems VV&T

To cope with increasing demands in processing power of embedded real-time system for controlling complex systems, multi-core architectures are the next generation priority in ERT software implementation and testing. It is sometime become impossible to satisfactorily validate a multiprocessor dynamic system algorithm with real-world scenario.

Real-world modeling of sensors is a challenging task because of performance degradation due to ageing, quantization, vibration, noise and nonlinearities, etc. Verification of controllable behavior of ERT control system in sensor fault/failure case is hard to perceive. Reliability of actuators is also a main factor to ensure the fast and precise performance of an embedded control algorithm. Actuating devices are sometime not possible to verify in complete system configuration because of their cost and complexity for example aircraft's thrust vectoring system is normally verified at component level.

On-time project completion with reliable VV&T procedures is difficult using tradition methodologies. Verification and validation of In-Vehicle control systems is an open challenge for engineers due to increasing demand of safety, comfort, performance, and sustainability [22]. Effective communication between software developers and VV&T engineers is a problem because of varying development platforms, concepts, terminologies, and acronyms. So, it is preferable to use single software development platform for all major activities from design, development to VV&T i.e., Matlab/Simulink and Labview.

Environmental effects, uncertainties, modeling errors, and testing equipment limitations are main hurdles in effective VV&T. Cost and resource requirements for modeling and real-time simulation development is also a challenge for reliable VV&T and also not widely shared because of the projects sensitivity. Rigorous and repeatable methods are required for modeling, simulation, testing, formal assessment and qualitative assessment. The challenge is how to ensure that best optimized practices are employed.

Autonomous, intelligent and adaptive ERT control algorithms VV&T is a major challenge for uninhabited aerial vehicles (UAVs), automobiles, aircrafts, satellite vehicles, and space robots certification [23–27].

5 Case Study in VV&T of Aerospace System

In industry, VV&T is incorporated as a standard procedure in the product design cycle. Here, one of the most challenging industrial application is discussed in detail. Hardware in the loop testing of an autonomous aerial vehicle is presented in the next section.

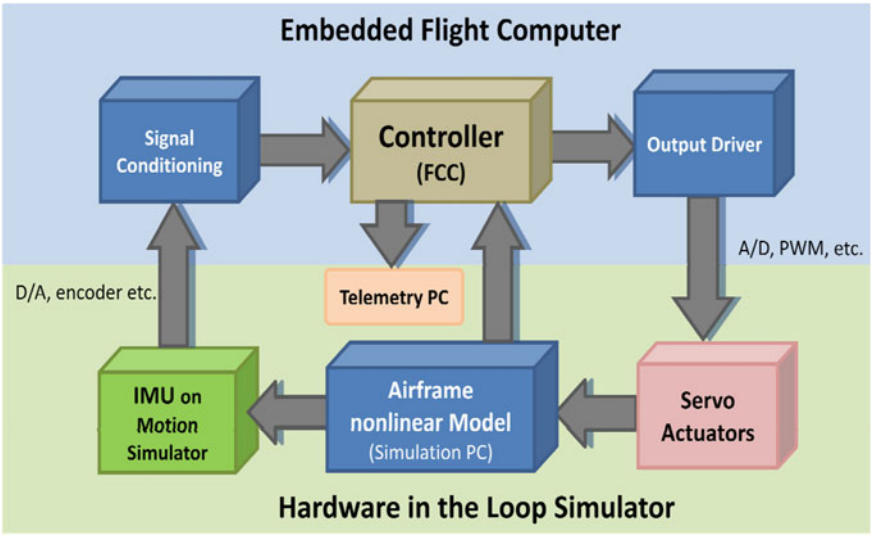


Fig. 19 Hardware in the loop testing of UAV avionics

5.1 VV&T of a Modern Autonomous UAV System

An unmanned aerial vehicle is an autonomous embedded platform with many sensors, actuators and multiprocessor systems where critical subsystems are responsible for overall system performance. Hardware in the loop test setup is a part of VV&T procedure as shown in the Fig. 14. The modern day flight vehicles are intelligent systems used for various applications where survivability and mission completion is one of the prime objectives. Figure 19 depicts the general layout of the HILS testing for the verification of flight computer code and sensor/actuator integrity. Airframe model with nonlinear environment, disturbances and wind gust model as well as the flight profile is simulated in the Simulation PC. The six degrees of freedom (6-DOF) motion simulator actually simulates the airframe with its inner axis, middle and outer axis each corresponding to the roll, pitch and yaw of the UAV system. Thus, the attitude of the UAV in the 3 axis as shown in the graphical interface on the simulation PC is closely followed by the motion simulator. The flight control computer (FCC) gives the actuator commands which are sensed by the nonlinear simulation model. The IMU senses the corresponding roll/pitch/yaw of the UAV and sends these measurements to the FCC which takes appropriate action to ensure the attitude as per flight plan. In addition, the telemetry PC shows all the important flight parameters to the user.

Two real-time test benches are setup using two separate technologies. One solution is based on Matlab real-time windows target (RTWT) with graphical autopilot interface and other one based on start-of-the-art dSPACE real-time interface (RTI) with ControlDesk technology for the flight testing of customized commercial

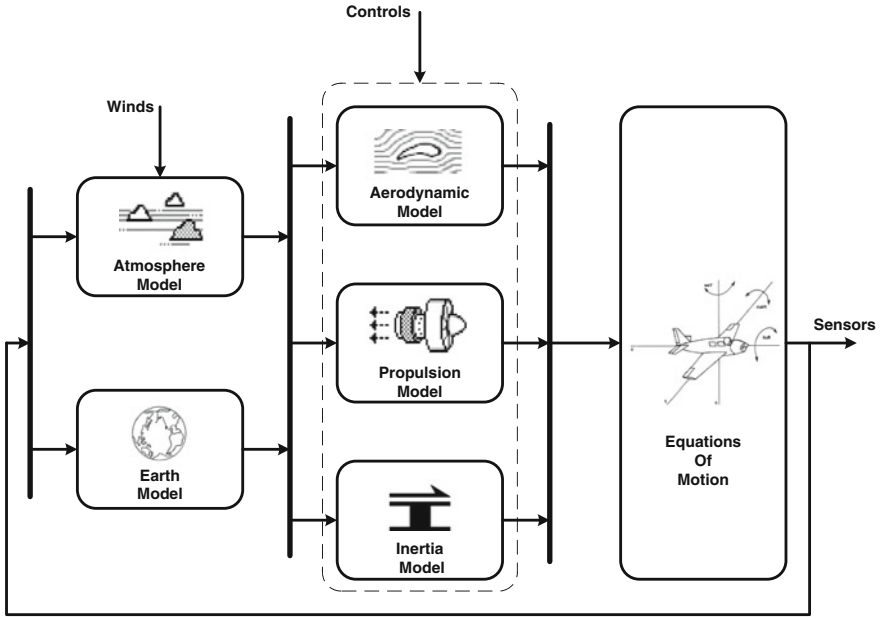


Fig. 20 Basic layout of AeroSim non-linear aircraft

of-the-self (COST) UAV system. A low cost off-the-shelf autopilot and sensor system is used to develop this platform.

Here the plant model is a physical, functional, logical, and mathematical description of an aircraft and it's environment which replicates the real complex system characteristics using data fitting, system identification, physical modeling, parameter estimation and first-principles modeling techniques. Most of the real-world systems are highly nonlinear and their respective models can be developed by expressing them in high-order complex mathematical terminologies to increase their accuracy. The developed models are not 100 % accurate with respect to the true system; however, they are quite useful for understanding and sufficient for controlling the system.

UAV nonlinear simulation model is modified from Aerosonde UAV Simulink model available in AeroSim Blockset from Unmanned Dynamics [28]. The AeroSim Blockset and Aerospace Blockset library provide almost all the aircraft model components, environment models, and earth models for rapid development of 6-DOF aircraft dynamic models. Figure 20 shows a basic layout of nonlinear aircraft model subsystems from AeroSim Blockset library [28]. Customization of the aerodynamics, propulsion, and inertia Aerosonde UAV simulink models is performed after extensive experimentation on our small UAV. Generally, the equation of motion, earth and atmosphere Simulink models are not modified because they are independent of the aircraft system used. Brief description of adapted aircraft subsystem models is presented in next section. First order actuator dynamics with saturation limits are used to simulate the control surface characteristics.

Aerodynamic model of a 6-DOF conventional aircraft is usually derived from equations of X, Y, and Z forces acting on the vehicle body x, y, and z direction and L, M, and N moments acting on vehicle body x, y, and z direction [29]. Force equations used can be found in [29] and they are expressed below. The summation of the forces in each body axis gives linear velocity state equations.

$$\dot{u} = rv - qw - g_0' \sin\theta + \frac{F_x}{m} \tag{1}$$

$$\dot{v} = pw - ru + g_0' \sin\phi\cos\theta + \frac{F_y}{m} \tag{2}$$

$$\dot{w} = qu - pv + g_0' \cos\phi\cos\theta + \frac{F_z}{m} \tag{3}$$

where u, v, and w are the body axis linear velocities in m/s, p, q, and r are the body axis angular rates in rad/s, ϕ, θ, and ψ are the body attitude angles, F_x, F_y, and F_z are the force components in each body axis.

Moment Equations are derived by considering moment about the aerodynamics center of the aircraft, the 6-DOF moment equations are given as

$$\dot{p} = (c_1 r + c_2 p)q + c_3 L + c_4 N \tag{4}$$

$$\dot{q} = c_5 pr - c_6(p^2 - r^2) + c_7 M \tag{5}$$

$$\dot{r} = (c_8 p - c_2 r)q + c_4 L + c_9 N \tag{6}$$

where c_1–c_9 are the interia coefficients which are computed using AeroSim library. The moments L, M, and N include all the available loads (i.e. aerodynamics, thrust, winds) and they are given with respect to the current location of aircraft center of gravity (CG).

The kinematic equations of the aircraft rotation motion using classical Euler angles ϕ, θ, and ψ with the body angular rates p, q, and r and aerodynamics angles α, β, and γ are given by

$$\dot{\phi} = p + \tan\theta(q\sin\phi + r\cos\phi) \tag{7}$$

$$\dot{\theta} = q\cos\phi - r\sin\phi \tag{8}$$

$$\dot{\psi} = \frac{q\sin\phi + r\cos\phi}{\cos\theta} \tag{9}$$

$$\theta = \gamma + \alpha\cos\phi + \beta\sin\phi \tag{10}$$

The **propulsion system** models the interaction between the electric motor and propeller dynamics. The electrical characteristics of motor is used to describe the motor dynamics and rotation speed of propeller (ω_p) is used to describe the propulsion dynamics. Our small aircraft flight dynamics are sensitive to propulsion dynamics because of the large torque from the propulsion system as relative to its size. The propulsion system dynamics is expressed in the following equation using conservation of angular momentum.

$$(I_{motor} + I_{propeller})\dot{\omega}_p = T_{motor} - T_{propeller} \tag{11}$$

where I_{motor} and $I_{propeller}$ are the moment of inertia of rotating motor body and propeller respectively in kgm^2, T_{motor} and $T_{propeller}$ are the output torque at motor shaft and torque generated by propeller respectively in Nm.

The **inertia model** consists of the aircraft inertia parameters, including mass, CG position, and moment of inertia coefficients. The aircraft moment of inertia is described by the moment of inertia matrix I as

$$I = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & -I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix}$$

Beside the aircraft moment of inertia matrix, propulsion system moment of inertia coefficients are determined using pendulum method in lab experimentation setup.

The linearized state-space aircraft model obtained at trim point in desired flight envelope using numerical linearization is given by:

$$\dot{x} = Ax + Bu \tag{12}$$

$$y = Cx + Du \tag{13}$$

where x is state vector consists of $[\phi\, \theta\, \psi\, p\, q\, r\, u\, v\, w\, h\, \omega_p]^T$ where ϕ, θ, and ψ are the Euler angles in rad, p, q, and r are body axis angular rates in rad/s, u, v, and w are the body axis velocities in m/s, h is the altitude in m and ω_p is the propeller speed in rad/s. The control input vector u consists of elevator, aileron, rudder and throttle inputs given by $[\delta_e\, \delta_a\, \delta_r\, \delta_t]^T$ in rad. y is the output vector consists of $[V\, \beta\, \alpha\, \phi\, \theta\, \psi\, h]^T$ where V is the airspeed in m/s, β is the sideslip angle in rad and α is the angle of attack (AOA) in rad. The properties of the system are expressed in state matrix A $(n \times n)$ and input matrix B $(n \times r)$ that weight the inputs. The output equation matrices C and D are determined by the particular choice of output variables.

The flight control algorithm is based on classical Proportional-Integral-Derivative (PID) controller tuned at trimmed flight conditions. Figure 21 shows the complete UAV model-based flight control loop with pilot commands input and real-time flight

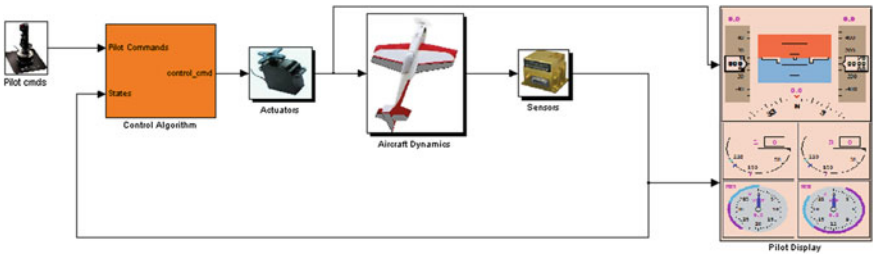


Fig. 21 UAV flight control simulation loop in Simulink

status display in Simulink. Initially, control algorithm is implemented in C S-function form and then executed on 32-bit MPC565 embedded microprocessor running at 56 MHz using Simulink embedded target code generation technology. Flight code is first verified with dummy control surfaces on actual actuators using digital-to-analog converters (DAC) and analog-to-digital converters (ADC) of NI6025E DAQ card. Afterward simulated sensors are replaced by available real sensors using high speed Quatech RS232/RS422 serial interface in HILS.

Autopilot flight control computer has embedded hardware which can be completely programmed through Simulink using Embedded Target toolbox for Motorola PowerPC (ETMPC) and Real-Time Workshop Embedded Coder (RTWEC). This high performance embedded solution is equipped with floating point unit (FPU), third generation time processor unit (TPU), 1 MB of Flash memory and queued serial multichannel modules (QSMCM) that offer highspeed UART and SPI functionality. Flight control algorithm based on CS-function is replaced by actual flight control processing hardware for PIL simulation testing and verification.

Matlab/Simulink provides modeling and simulation of UAV flight control loop with external mode to communicate with embedded hardware and nonlinear plant components in real-time windows environment. Figure 22a shows the bank angle (ϕ) tracking of UAV with roll rate (p), pitch rate (q), and yaw rate (r) state variables in real-time (RT) and HIL simulation cases. Commanded actuator deflection with feedback data acquisition through NI6025e connected in HIL simulation setup is shown in Fig. 22b.

In Fig. 22 RT simulation means that plant model (containing flight dynamics, actuators, and sensors) and Controller model (in terms of discrete time PID controller equations) both are running on the simulator processor (PowerPC 750GX) in Hard RT environment also known as 'software in loop simulation' (SIL). While, HIL simulation means that actual UAV flight controller based on powerPC processor (MPC565) will come into loop with other actual flight components like pilot stick, actuators and sensors. Only simulation RT data acquisition and flight simulation online comparison is performed through simulator's processor in HIL simulation for further analysis and V&V purposes.

Aircraft flight simulation avionics interface is developed using Gauges Blockset library as shown in Fig. 23. It includes airspeed indicator, artificial horizon, altimeter, turn coordinator, horizontal situation indicator, and a compass. UAV nonlinear 6-DOF simulation model consists of full nonlinear equations of motion. Aircraft actuator subsystem consists of nonlinear actuators model with rate-limiter and position-limiter. Sensor subsystem consists of sensor dynamics with noise and disturbance model. Complete simulation is running at 20 ms sample time thread and flight controller is executed at 40 ms sample time thread.

Flight data telemetry is performed on-board using high speed RS422 link with data checksum capability at 20 ms sampling rate. RF telemetry is also carry out for remote online flight data display and diagnostics in case of accident at 40 ms. HIL simulation is the extension of PIL and SIL simulations that includes actual actuators with dummy control surfaces and sensors on a 6-DOF motion platform for replicating the aircraft motion during flight. Actuators motion is also monitored

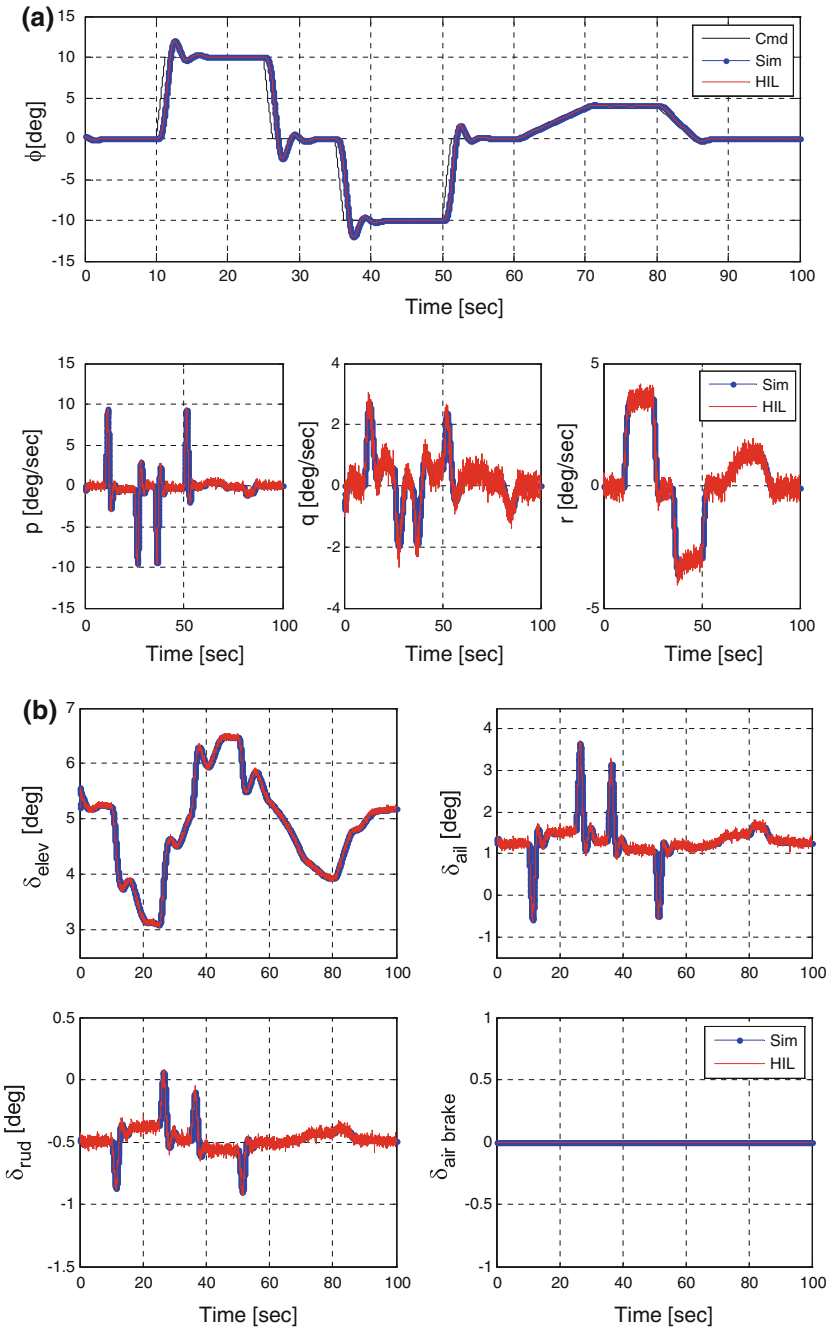


Fig. 22 RT and HIL simulation responses. **a** RT and HIL simulation response in tracking bank angle command (ϕ) with states, **b** Comparison of RT and HIL simulation control surfaces deflections

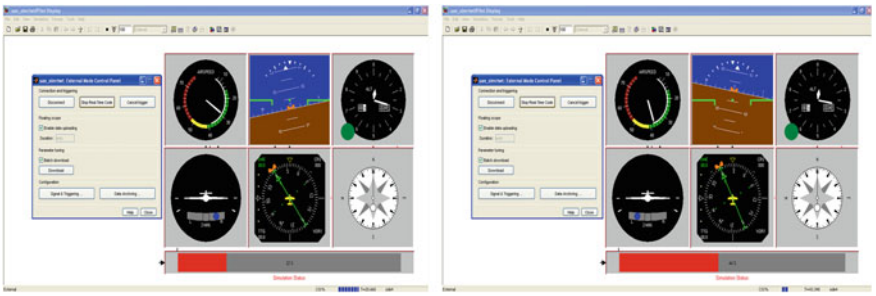


Fig. 23 UAV real-time avionics display at different flight conditions in RTWT

for troubleshooting any problem in FCC command implementation. Pilot command input is also rate and position limited to avoid improper command which can cause damage to the system.

Model-based UAV system development in Matlab/Simulink helps to reduce the risk, developmental costs, and time. HIL simulation offers complete system testing, verification, and validation of UAV on ground with actual flight components in hard real-time environment. Further, advantages include;

- Rapid testing and identification of FCS issues before real flight test.
- Multiple simulation runs are possible to verify the accuracy and repeatability of system performance.
- Provides a test bed for different flight controllers performance testing and comparison. Fault injection mechanism is also possible to test the fault-tolerance capability of the FCS in closed-loop environment.
- Provides a close to real environment for the pilot and flight test engineers to feel the actual flight situations.
- It can be used for each individual subsystem testing, verification and validation by simulating the other system components in real-time environment.
- It can be used for post-flight analysis and test flight data verification.

Initially, PID controllers are implemented for each longitudinal and lateral mode of UAV flight development and testing. Further work includes optimal controller design, adaptive controller design and inclusion of actuators fault-tolerance using control allocation (CA) strategies as detailed in [30, 31]. Also a fault detection and isolation (FDI) block can be introduced in modular flight controller design to handle actuators and sensors faults. FlightGear a free open-source flight simulator can also be you to visualize the aircraft flight motions.

Another HIL simulation setup is developed with Matlab/Simulink model-based environment using dSPACE hard real-time software development kit and dedicated hardware. It is not as cost effective solution as the former one but has superior real-time performance with on-line data display and precise high-speed data storage. For hard real-time performance testing, verification, and validation dSPACE systems are

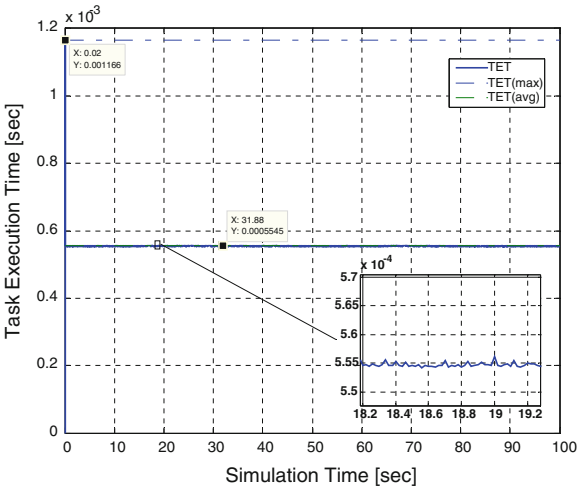


Fig. 24 Aircraft real-time simulation task execution time (TET) at each sample instant using DS1005PPC board

generally preferable. Here, we use Matlab2008b with ControlDesk 3.3 for the development of HIL simulation platform for UAV flight controller VV&T. UAV flight simulation task execution time (TET) in dSPACE is quite impressive as shown in Fig. 24. Worse case time required to complete real-time task in dSPACE is approximately 1.2 ms and on average approximately 560 μs are required to compete different tasks in complete 100 s UAV flight controlled simulation.

We used modular processor board DS1005 PPC for real-time execution of simulation which consists of PowerPC 750GX processor running at 933 MHz having 128 MB SDRAM, 16 MB flash memory, and two general-purpose timers. High speed PHS bus interface is used to communicate with other modular DAQ cards. DS2201 modular I/O board is used for DAC, ADC and DIO requirements. DS4201-S high speed serial interface board is used for RS232 and RS422 communication for sensors and telemetry data saving from FCC. dSPACE's TargetLink code generation technology can generate highly readable and optimized code for resource-limited embedded real-time systems. Graphical user interface is also developed in ControlDesk to visualize the flight data and status as shown in Fig. 25.

6 Solutions and Recommendations

Model-based software tools: Model-based real-time software development and VV&T techniques are now become market demand because of reliability, flexibility, and fast solution. Here we present some of the model-based embedded real-time systems design tools available in Matlab/Simulink® which is now become an

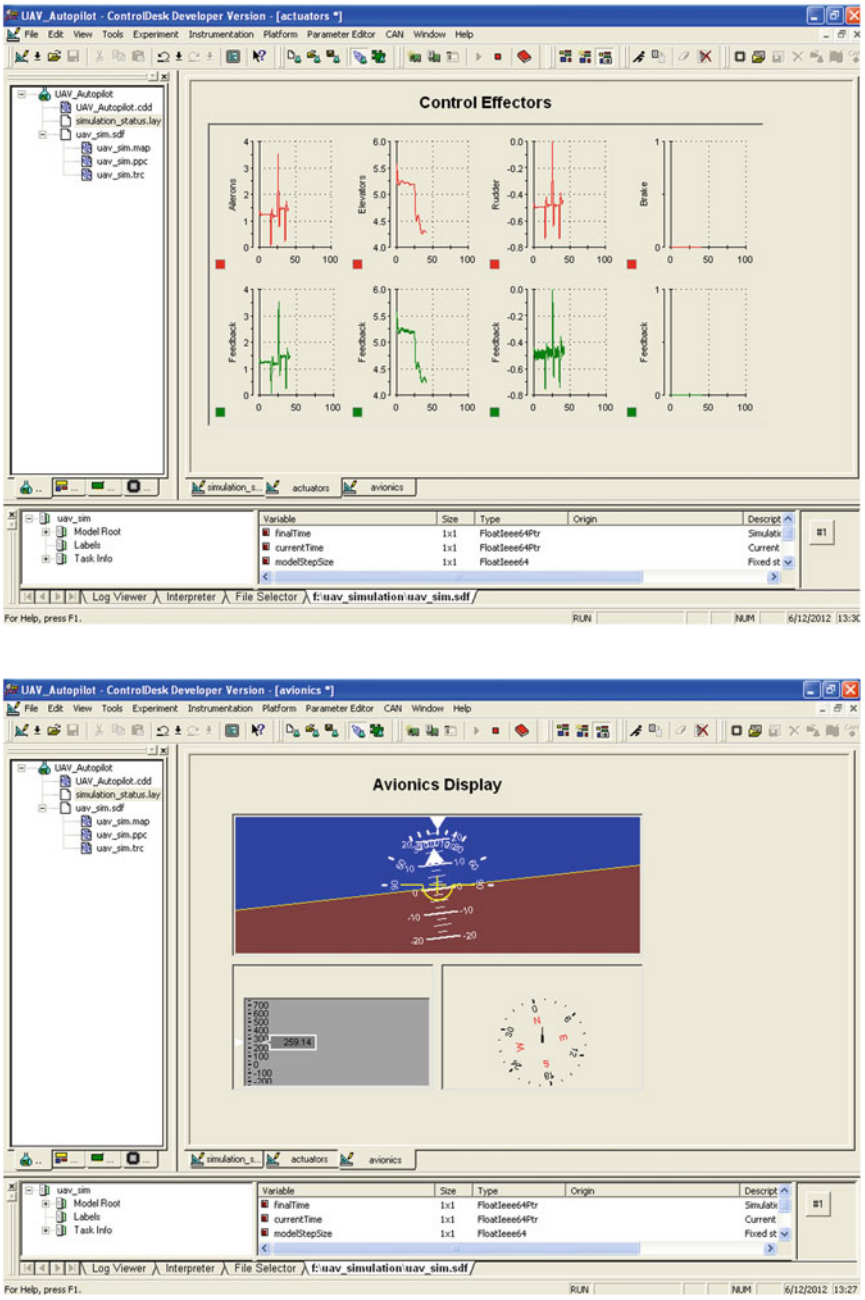


Fig. 25 Real-time simulation graphical user interface in controlDesk. **a** UAV real-time control effectors deflections [deg] layout in controlDesk, **b** UAV real-time avionics display layout in controlDesk

educational and industrial standard for complete system design [32]. There are various hardware/software solutions available within Matlab/Simulink® for real-time embedded system development and its verification, validation and testing. Following are the three ways to prepare complete embedded RT system.

Using Matlab/Simulink® family of products: There are following Matlab/Simulink family of products available for model-based verification, validation and testing in safety-critical product design:

1. Simulink: Simulation and model-based system design.
2. Simulink Coder: Generate C and C++ code from simulink and stateflow model.
3. Embedded Coder: Generate optimized embedded C and C++ code.
4. Simulink Verification and Validation: Use for simulink models and generated codes verification.
5. Simulink Code Inspector: Source code review according to DO-178 safety standards.
6. Simulink Design Verifier: Verify design according to requirements and generate test vectors.
7. Real-Time Workshop: Generate simplified, optimized, portable, and customizable C code from Simulink model-based design.
8. Real-Time Windows Target: Execute Simulink models in real time on Win OS based PC.
9. xPC Target: Use for real-time rapid prototyping and HIL simulation on PC hardware.
10. xPC TargetBox: Embedded industrial PC for real-time rapid prototyping.
11. Real-Time Workshop Embedded Coder: Embedded real-time code generator for product deployment.

Using Hard Real-Time Operating Systems (RTOS): In this approach, we develop our system model and simulation in Matlab/Simulink® and generate C code with some modifications for porting to RTOS. Then we run generated code in RTOS with customized graphical user interface (GUI) with some necessary improvements. For detailed procedure as a reference see [33]. Various free and commercial hard real-time OS available for this type of embedded hardware and software VV&T. Advantages of this procedure are customized multiple user interfaces, thousands of available hardware I/O interfaces and hard real-time verification and validation utilizing maximum hardware performance.

Using Commerical-Of-The-Shelf (COTS) Solutions: Several complete improved rapid prototyping hardware and GUI development solutions are available with Simulink model-based system design. Some of them are as follows:

1. dSPACE Systems Inc. (ControlDesk, dSPACE simulators, dSPACE RapidPro, and TargetLink)
2. OPAL-RT Technologies, Inc. (RT-LAB, RT simulators (eMEGAsim, eDRIVEsim and eFLYsim), RCP controllers)
3. Applied Dynamics International (Advantage Framework (SIL and HIL simulation environment), Beacon family (embedded code generator) and Emul8 family (rapid prototyping environment).

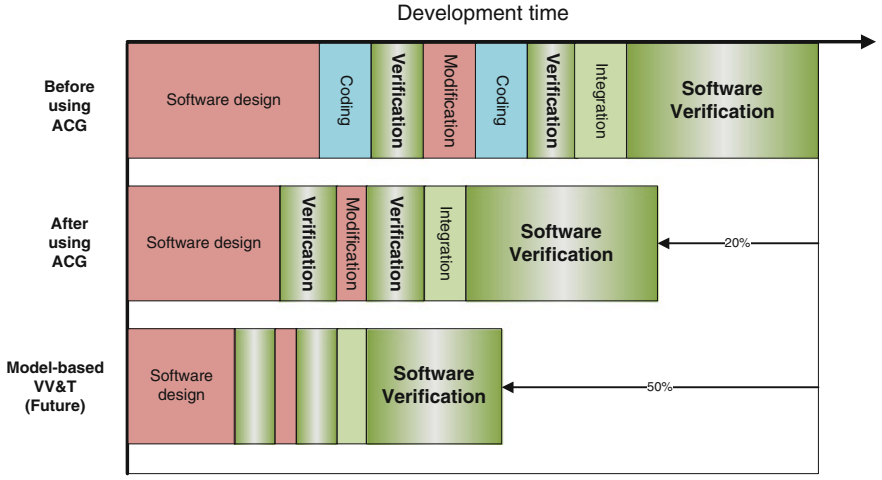


Fig. 26 Product development time reduction using model based design (Ref. [22])

4. Quanser Inc. (Real-time control software design, implementation and rapid prototyping tools for algorithm VV&T)
5. NI LabVIEW real-time toolkits
6. Humsoft (Real-time toolbox and Data acquisition boards for VV&T)
7. TeraSoft Inc. (HIL and RCP hardware solutions for VV&T)
8. UEI Inc. (Real-time HIL and RCP hardware solutions)
9. DSPtechnology Co. Ltd (RTSim (HIL and RCP hardware and software))
10. Pathway Technologies Inc. (RCP electronic control units for software VV&T)
11. add2 Ltd (RT HIL and RCP hardware and software for algorithm VV&T)

etc. . .

These model-based verification and validation solutions are cost effective, reliable and fast as compare to other tradition solutions. In embedded real-time product design VV&T takes 40–50 % project time which can easily be reduce utilizing model-based automatic code generation (ACG), verification, validation, and testing techniques as shown in Fig. 26 by taking an example of automobile industry. All above ways for the development and VV&T of ERT software made Matlab/Simulink a promising candidate for increasing productivity and reducing project cost. Modular "off the self" hardware platforms availability provide maximum flexibility for cost, selection of communication interfaces, and performance requirements.

7 Conclusion

Verification, Validation and Testing technologies are important for next generation safety critical systems. This chapter has described ongoing work in applying and extending COTS, specifically HIL simulations, to the VV&T of an embedded

real-time software product. In addition, a detailed overview of methods followed in verification, validation and testing are also listed. Verification and Validation of the embedded software through V-model using Matlab/Simulink is found to successfully present the product life cycle model. Hardware in the loop (HIL) testing plays an important role in V-model by physically connecting the embedded controller, sensor and actuators in the closed loop to verify the data integrity, interfacing and decision logic of the controller based on the sensor information. Two case studies are used to demonstrate the underlying concepts in VV&T and its practical implementation.

References

1. J.A. Stankovic, Misconceptions about real-time computing: a serious problem for next-generation systems. Computer **21**(10), 10–19 (1988)
2. H. Kopetz, *Real-Time Systems Design Principles for Distributed Embedded Applications* (Kluwer Academic Publishers, London, 1997)
3. C.D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*. Technical Report (CMUCS-86-134 Carnegie-Mellon University, Department of Computer Science, USA, 1986)
4. M. Grindal, B. Lindström, Challenges in testing real-time systems. Presented at in 10th international conferene on software testing analysis ad review (eurostar' 02), Edinburgh, Scotland, 2002
5. J.W.S. Liu, *Real-Time Systems* (Prentice Hall, New Jersey, 2000)
6. VDC Research, *Next Generation Embedded Hardware Architectures: driving Onset of Project Delays, Costs Overruns, and Software Development Challenges*. Technical report, Sept 2010
7. M. van Genuchten, Why is software late? An empirical study of reasons for delay in software development. IEEE Trans. Softw. Eng. **17**(6), 582–590 (1991)
8. El al flight 1862, *Aircraft Accident Report 92–11*. Technical report (Netherlands Aviation Safety Board, Hoofddorp, 1994)
9. IEEE Standard 610.12-1990, *Standard Glossary of Software Engineering Terminology* (IEEE Service Center, NY, 1990)
10. I. Sommerville, *Software Engineering*, 6th edn. (Addison-Wesley Publishing Company, MA, 2001)
11. W. W. Royce, Managing the development of large software systems. Proceedings of Western Electronic Show and Convention, pp. 1–9, 1970. Reprinted in Proceedings of the 9th International Conference on, Software Engineering, pp. 328–338, 1987
12. C. Kaner, J. Falk, H. Nguyen, *Testing Computer Software*, 2nd edn. (Van Nostrand Reinhold, NY, 1999)
13. R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools* (Addison-Wesley, MA, 1999)
14. IEEE Standard 1028–1988, *IEEE Standard for Software Reviews* (IEEE Service Center, NY, 1988)
15. *Simulink Verification and Validation, User's Guide*, Mathworks, Inc., http://www.mathworks.com
16. *Matlab and Simulink Mathworks*. http://www.mathworks.com
17. J.A. Whittaker, What is software testing? And why is it so hard? IEEE Softw. **17**(1), 70–79 (2000)
18. J. Wegener, M. Grochtmann, Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Syst. **15**(3), 275–298 (1998)
19. J. Hänsel, D. Rose, P. Herber, S. Glesner, *An Evolutionary Algorithm for the Generation of Timed Test Traces for Embedded Real-Time Systems*. IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), 2011, pp. 170–179

20. R.L. Haupt, S.E. Haupt, *Practical Genetic Algorithms* (Wiley, New York, 2004)
21. B. Beizer, *Software Testing Techniques*, 2nd edn. (VNR, New York, 1990)
22. Toyota, North America Environmental Report (Toyota Motor North America, Inc., NY, 2010).
23. S.A. Jacklin, J. Schumann, P. Gupta, K. Havelund, J. Bosworth, E. Zavala, K. Hayhurst, C. Belcastro, C. Belcastro, *Verification, Validation and Certification Challenges for Adaptive Flight-Critical Control Systems* (AIAA Guidance, Navigation and Control, Invited Session Proposal Packet, 2004)
24. L. Pedersen, D. Kortenkamp, D. Wettergreen, I. Nourbakhsh, A survey of space robotics. Robotics (2003)
25. N. Nguyen, S.A. Jacklin, *Neural Net Adaptive Flight Control Stability, Verification and Validation Challenges, and Future Research* (IJCNN Conference, Orland Florida, 2007)
26. J.M. Buffington, V. Crum, B. Krogh, C. Plaisted, R. Prasanth, *Verification and Validation of Intelligent and Adaptive Control Systems, in 2nd AIAA Unmanned Unlimited Systems Conference* (San Diego, CA, 2003)
27. J. Schumann, W. Visser, Autonomy software: V&V challenges and characteristics, in Proceedings of the 2006. IEEE Aerospace Conference, 2006
28. Unmanned Dynamics LLC. Aerosim Blockset Version 1.2 User's Guide, 2003
29. B.L. Stevens, F.L. Lewis, *Aircraft Control and Simulation* (John Wiley & Sons, Inc., 1992). ISBN 0-471-61397
30. A.H. Khan, Z. Weiguo, Z.H. Khan, S. Jingping, Evolutionary computing based modular control design for aircraft with redundant effectors. Procedia Eng. **29**, 110–117 (2012). (2012 International Workshop on Information and Electronics Engineering)
31. A.H. Khan, Z. Weiguo, S. Jingping, Z.H. Khan, Optimized reconfigurable modular flight control design using swarm intelligence. Procedia Eng. **24**, 621–628 (2011). (International Conference on Advances in Engineering 2011)
32. *MathWorks User Stories*. http://www.mathworks.com/company/user_stories/index.html
33. N. ur Rehman, A.H. Khan, RT-Linux Based Simulator for Hardware-in-the Loop Simulations. International Bhurban Conference on Applied Sciences and Technology (IBCAST), Islamabad, 2007, pp. 78–81

A Multi-objective Framework for Characterization of Software Specifications

Muhammad Rashid and Bernard Pottier

Abstract The complexity of embedded systems is exploding into two interrelated but independently growing directions: architecture complexity and application complexity. Consequently, application characterization under the real input conditions is becoming more and more important. State-of-the-art application characterization frameworks mainly focus on a single design objective. A general purpose framework is required to satisfy multiple objectives of early design space exploration. This chapter proposes a multi-objective application characterization framework based on a visitor design pattern. High level source specifications are transformed into a trace tree representation by dynamic analysis. Trace tree representation is analysed by using visitor design pattern to get run-time characteristics of the application. Among other outcomes, application orientation and inherited spatial parallelism are key concerns in this article. Experimental results with MPEG-2 video decoder shows viability of the proposed framework.

1 Introduction

Software specifications of multimedia standards are complex and it is virtually impossible to analyse these applications without generic automated tools [1, 2]. As a result, architectural implementation choices based on merely designer experience without objective measures become extremely difficult or impossible task and may lead to costly re-design loops. Measuring application complexity without any architecture directives is critical at the beginning of a design cycle. The process of measuring application complexity at early stages of the design flow is called application

M. Rashid (✉)
Umm Al-Qura University, Makkah, Saudi Arabia
e-mail: mfelahi@uqu.edu.sa

M. Rashid · B. Pottier
University of Bretagne Occidentale, CNRS, UMR 3192, Lab-STICC, Brest, France

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_7,

characterization [3]. The objective of application characterization is to effectively map complex applications on heterogeneous multi-core architectures.

The performance gain in multi-core architectures depends upon application parallelization across the cores. Sequential languages such as C, C++, Smalltalk are portable, high performance and maintainable for uni-processors. However, an efficient implementation on multi-core platforms raises two key challenges: (a) parallel tasks must be extracted from sequential specifications and (b) there must be an excellent match between extracted tasks and architecture resources. Any significant mismatch results in performance loss and a decrease of resource utilization [4, 5]. Therefore, application characterization through dynamic analysis is very important at the beginning of design cycle.

In existing application characterization frameworks [6–15], analysis results are restricted to a single design objective such as application affinity, communication bandwidth, data level parallelism, task level parallelism, spatial parallelism and so on. However, a general purpose application characterization framework at early stages of the design cycle is a critical requirement.

In this article, we propose a general purpose application characterization framework for early design space exploration. The application is written in an object oriented language Smalltalk [16] and is automatically transformed into a trace tree representation by dynamic analysis [17]. The trace tree represents implementation-independent specification characteristics. It provides information about the inherent characteristics of the application. We present a generic analysis approach to analyse the trace tree representation for design space exploration.

The exploration of design space for embedded systems involves multiple analysis requirements [18]. The proposed framework is generic such that it can extract various application characteristics depending upon a particular analysis requirement by simply defining new analysis operations on the trace tree representation. In this article, we focus on extraction of spatial parallelism as well as application orientation in terms of processing, control and memory accesses.

In order to highlight the significance of analysis results, spatial parallelism information is used to describe sequential application in the form of parallel process networks. For this purpose, we propose *AVEL* framework that exposes inherent parallelism and communication topology of the application similar to streaming programming languages [19]. The *AVEL* processes are abstract programmable units and implemented with behavioral code of Smalltalk. Experimental results with MPEG-2 video decoder [20] proves viability of the proposed framework.

This article is organized as follows: Sect. 2 describes state-of-the-art in: application analysis techniques, application characterization and streaming languages. Section 3 provides essential background knowledge to understand the contents in subsequent sections. Section 4 proposes a general purpose dynamic analysis framework to extract run-time characteristics of the application. Section 5 presents some usage scenarios of analysis results. Section 6 provides experimental results with MPEG-2 video decoding application. Section 7 describes the application in the form of parallel process networks. Finally, Sect. 8 concludes the article.

2 Review of Related Work

We have divided our related work into three categories: application analysis techniques, application characterization and streaming programming languages.

2.1 Application Analysis Techniques

The purpose of presenting application analysis techniques is to provide the background for state-of-the-art application characterization approaches described in Sect. 2.2. Application analysis techniques are further sub-divided into two types: static analysis and dynamic analysis.

In static analysis, we are interested in Worst Case Execution Times (WCET) which are in general very difficult and even impossible to compute, hence the necessity to develop methods for WCET estimation that are fast to compute and safe [21]. Several techniques for WCET estimation have been proposed. Wihelm et al. has presented an excellent review of existing techniques [22].

Drawback of static analysis is that it can only detect upper and lower bounds while the processing complexity of multimedia algorithms heavily depends on the input data statistics. Microarchitecture of the modern microprocessors is so complex with caches, pipelines, pre-fetch queues and branch prediction units that accurately predicting their effects in execution time is a real challenge. More specifically, static analysis techniques attempt to determine bounds on the execution times of different tasks when executed on a particular hardware while the objective in this article is to determine the algorithmic characteristics without any architecture declaratives.

A systematic survey of program comprehension through dynamic analysis is presented in [17]. In dynamic analysis, code instrumentation is performed by modifying the source code. Code instrumentation is the process of inserting additional instructions or probes into the source code. The limitation of source code modification technique is that all instrumented functions have to be re-parsed and re-compiled [23]. Another re-compilation may be needed to reinstall the original functions.

Debugger and instruction-level profiling are also the forms of dynamic analysis. Although, debugger does not modify the source code, the disadvantage is the considerable deceleration of the system execution. Instruction-level profiling provides information at function level. The information gathered with profilers strictly depends on the underlying machine and on the compiler optimizations. This is against the requirement of high level system design in which complexity evaluation depends only on the algorithm itself.

Richner et al. have also presented an example of a trace representation by dynamic analysis [24]. It extracts trace events representing the function calls during execution. However, the analysis goals should not restrict the type of information to be collected. The objective is to extract maximum dynamic data and then depending on the requirements of a particular analysis, extract and use an appropriate sub-set of the dynamic data.

Table 1 Comparison of application characterization techniques for design space exploration

Tools/Authors	Source specifications	Analysis techniques	Instrumentation	Objective
Abdi et al. [6]	C	Static/Dynamic	Source code	Prune out the design space
Silvano et al. [7]	SystemC	Static /Dynamic	Source code	HW/SW partitioning
Rul et al. [8]	C	Dynamic	Source code	Function level parallelism
SPRINT [9]	C	Dynamic	Source code	Function level parallelism
Commit [10]	StreamIt	Dynamic	Source code	Extraction of parallelism
LESTER [13]	C	Dynamic	Source code	Application characterization
Prihozhy et al. [14]	C	Dynamic	Source code	Application characterization

2.2 Application Characterization

In this section, we describe state-of-the-art application characterization frameworks based on static analysis, dynamic analysis or a combination of both. A comparison is presented in Table 1.

Abdi et al. have presented an application characterization approach based on dynamic profiling and static re-targeting [6]. Dynamic profiling provides inherit characteristics of the application by instrumenting and simulating the source specifications without any architecture directives. In the re-targeting stage, output of the dynamic profiling is coupled with the target characteristics. The design space is explored with the results that are accurate enough to prune out infeasible design alternatives.

The approach of **Silvano et al.** is based on a combination of static and dynamic analysis [7]. Application specified in a subset of systemC is first statically analyzed. It provides a set of data to express the affinity of the system functionalities towards a set of processing elements such as GPP, DSP and FPGA. After static analysis, dynamic analysis is performed to extract some run-time aspects of the application including profiling and communication cost. Finally, results from static and dynamic analysis are combined to estimate the load associated with each system functionality for HW/SW partitioning.

The primary goal of the work performed by Abdi et al. [6] and Silvano et al. [7] is to obtain application affinity metrics. These metrics are either dedicated to prune out the infeasible design space or HW/SW partitioning. They do not consider the extraction of spatial parallelism in their analysis.

A profile-based technique to extract parallelism from a sequential application is presented by Rul et al. [8]. It transforms source specifications to a graph-based representation for identifying parallel code. It measures memory dependencies between different functions of the application. Therefore, the granularity of extracted parallelism is larger and is not well-suited to extract fine grain parallelism.

SPRINT tool [9] generates an executable concurrent model in SystemC starting from C code and user defined directives. First, it transforms C code to control flow graph which is further transformed to model different concurrent tasks. Again, this tool only extracts function level parallelism and ignores fine grain parallelism. Moreover, no application orientation is provided to bridge the application-architecture gap.

Commit research group from Massachusetts Institute of Technology (MIT) presents a methodology to extract coarse grained pipeline parallelism in C programs [10]. The output is in the form of stream graph as well as a set of macros for parallelizing the programs and communicating the required data. However, the work in [10] does not provide any details about other forms of parallelism. The work is further expanded to expose task, data and pipeline parallelism present in an application written in a streaming programming language StreamIt [11].

LESTER research group has proposed a C based high level exploration methodology [12, 13]. The input application is transformed into an internal hierarchical graph based representation to compute some design metrics. These design metrics characterize the application in terms of computation operations, memory transfer operations, control operations and processing parallelism.

Ravasi and Mattavelli introduce an integrated tool for the complexity analysis of C descriptions in [23]. The tool is capable of measuring all C language operators during the execution of algorithms. The tool capabilities also include the simulation of virtual memory architectures, extending it to data transfer and storage analysis. It is extended to measure the parallelization potential of complex multimedia applications in [14, 15] by defining critical path metric.

The work of Ravasi and Mattavelli [14, 23] and LESTER [5, 13] take into account application characterization as well as the extraction of spatial parallelism. Despite their significant contributions in providing application-architecture mapping guidelines, these approaches are restricted to some special design metrics. These design metrics may not be able to fulfil some analysis requirements in a holistic design environment, such as to compute the amount of data transfer between two specific functions, or to compute the value of a specific variable in each step of the program execution.

2.3 Streaming Languages

We have described in the introductory part of this article that spatial parallelism information is used to describe the application in the form of parallel process networks. This section provides a brief overview of the streaming languages used for writing applications in the form of parallel process networks.

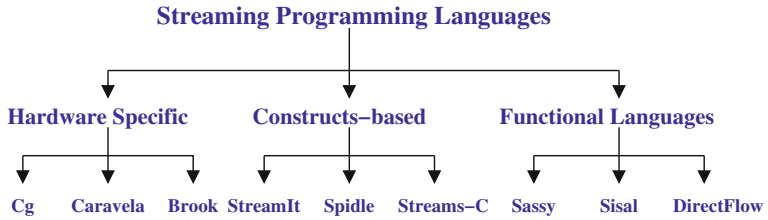


Fig. 1 Classification of streaming programming languages

The idea of language dedicated to stream processing is not new and has already been discussed in existing literature [25]. The languages of recent interests are **Cg** [26], **Brook** [27], **Caravela** [28], **StreamIt** [11], **Spidle** [29], **Streams-C** [30], **Sisal** [31], **Sassy** [32] and **DirectFlow** [33]. Existing stream languages can be divided into three categories as shown in Fig. 1.

The first type of languages are geared towards the features of specific hardware platform such as **Cg** [26], **Brook** [27] and **Caravela** [28]. All of these languages are dedicated to program Graphical Processing Units (GPUs). Cg language is a C-like language that extends and restricts C in certain areas to support the stream processing model. Brook abstracts the stream hardware as a co-processor to the host system. Kernel functions in Brook are similar to Cg shaders. These two languages do not support the distributed programming. Caravela applies stream computing to the distributed programming model.

The second type of languages introduce constructs for gluing components of stream library. The examples are **StreamIt** [11], **Spidle** [29] and **Streams-C** [30]. StreamIt and Spidle are stream languages with similar objectives. However, the former is more general purpose while the latter is more domain specific. StreamIt is a Java extension that provides some constructs to assemble stream components. Spidle, on the other hand, provides high level and declarative constructs for specifying streaming applications. Streams-C has a syntax similar to C and is used to define high level control and data flow in stream programs.

The third type of languages are functional languages such as **Sisal** [31], **Sassy** [32] and **DirectFlow** [33]. Sisal offers automatic exploitation of parallelism as a result of its functional semantics. Sassy is a single assignment of C language to enable compiler optimizations for parallel execution environments targeting particularly the reconfigurable systems. DirectFlow system is used for describing information flow in the case of distributed streaming applications.

Before describing the proposed framework, some background knowledge is required. Section 3 will provide essential background to understand the contents described in this article.

3 Background and Definitions

This section briefly reviews the basic concepts of an object oriented language Smalltalk. Then, the concept of visitor design pattern is illustrated. The proposed application analysis technique is based on visitor design pattern concept.

Smalltalk [16] is uniformly object-oriented because everything that the programmer deals with is an object, from a number to an entire application. It differs from most languages in that a program is not defined declaratively. Instead, a computation is defined by a set of objects. Classes capture shared structure among objects, but they themselves are objects, not declarations. The only way to add code to classes is to invoke methods upon them. Smalltalk is a reflective programming language because its classes inherently support self modifications.

Definition 7.1 *Reflective Programming*—the programming paradigm driven by reflection. The reflection is the process by which an application program observe and/or modify program execution at run-time. In other words, the emphasis of the reflective programming is dynamic program modification. For example, the instrumentation in dynamic analysis of application programs can be performed without re-compiling and re-linking the program.

Definition 7.2 *Visitor Design Pattern*—it represents an operation to be performed on the elements of an object structure [34]. It defines a new operation without changing the classes of the elements on which it operates. Its primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element objects".

The general organization of visitor design pattern is shown in Fig. 2. Abstract class for object structure is represented as AbstractElement while the abstract class for visitor hierarchy is represented as AbstractVisitor. "Visitor1" and "Visitor2" are inherited from abstract class. The functionality is simply extended by inheriting more and more visitor classes as each visitor class represents a specific function.

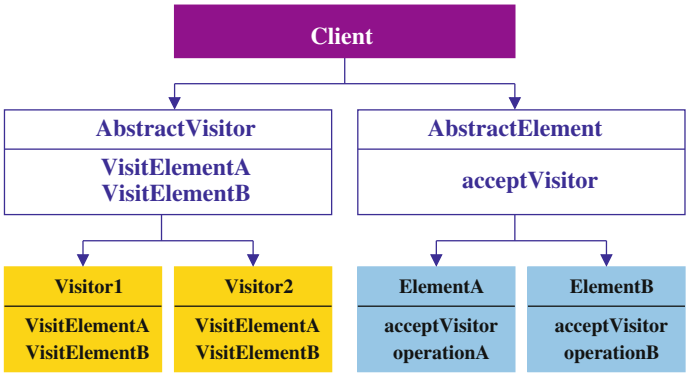


Fig. 2 General organization of visitor design pattern

3.1 Example of Visitor Design

Consider a compiler that parses an input program and represents the parsed program as an Abstract Syntax Tree (AST). An AST may have different kinds of nodes such that multiple operations can be performed on an individual node. Examples of nodes in an AST are assignment nodes, variable reference nodes, arithmetic expression nodes and so on. Examples of operations performed on an AST are program re-structuring, code instrumentation, computing various metrics and so on.

Operations on the AST treat each type of node differently. One way to do this is to define each operation in the specific node class. Adding new operations requires changes to all of the node classes. It can be confusing to have such a diverse set of operations in each node class. Another solution is to encapsulate a desired operation in a separate object, called as visitor. The visitor object then traverses the elements of the tree. When a tree node accepts the visitor, it invokes a method on the visitor that includes the node type as an argument. The visitor will then execute the operation for that node—the operation that used to be in the node class.

3.2 Uses and Benefits of Visitor Based Design

The visitors are typically used: (a) when many distinct and unrelated operations are performed on objects in an object structure and (b) when the classes defining the object structure rarely change and we want to define new operations over the structure. Visitor based design provides modularity such that adding new operations with visitors is easy. Related behavior is not spread over the classes defining the object structure. Visitor lets the designer to keep related operations together by defining them in one class. Unrelated sets of behavior are partitioned in their own visitor subclasses. The Smalltalk environment has a visitor class called as *ProgramNodeEnumerator*.

Definition 7.3 *ProgramNodeEnumerator*—an object to visit AST produced from the Smalltalk source code. The structure of AST is determined by the source code and the syntax rules of Smalltalk. Therefore, AST is also called as Program Node Tree. Consequently, *ProgramNodeEnumerator* class in Smalltalk environment provides a framework for recursively processing a Program Node Tree [34].

4 Application Analysis Framework

This section proposes an application analysis framework. The first part of this section describes application transformation into a trace tree representation (Sect. 4.1). The second part presents analysis of trace tree representation (Sect. 4.2).

Figure 3 shows the proposed application analysis framework. It is divided into two parts: The first part is related to transformation of Smalltalk source specifications into a trace tree representation by dynamic analysis. The second part is concerned with trace tree analysis to get desirable analysis results. The inputs to the first part

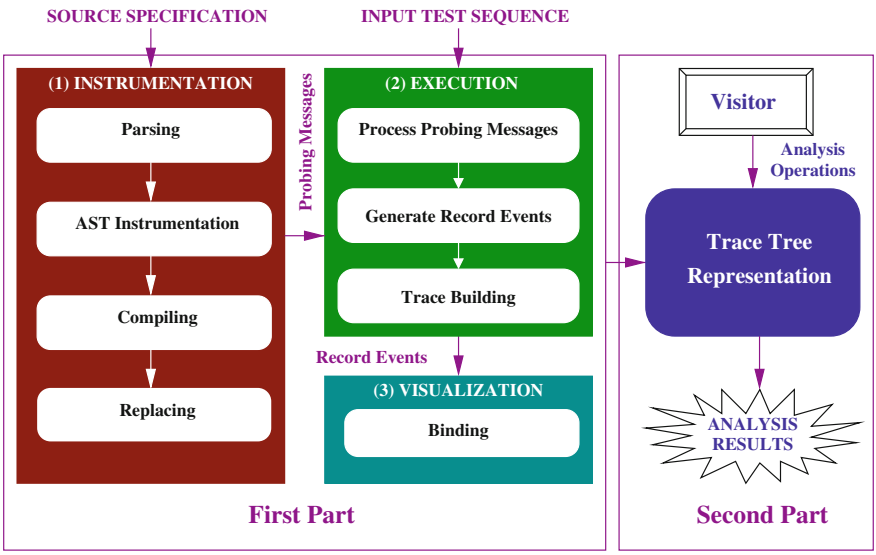


Fig. 3 Dynamic analysis framework for application characterization

are executable specifications along with real input data and the output is trace tree representation of input specifications. The input to the second part is trace tree and the output is in the form of analysis results.

4.1 Application Transformation into Trace Tree

This section describes the first part of proposed application analysis framework. It takes Smalltalk executable source specifications along with real input data. As the analysis is performed on executable specifications, therefore the input data is the real data and not merely synthetic vectors. For example, in case of MPEG-2 video decoding application analysis, the input is in the form of MPEG-2 video bit-stream. Dynamic analysis is performed to transform source specifications into a trace tree representation. The trace tree contains information about the execution of an application at run-time and represents implementation independent specification characteristics.

First part of Fig. 3 summarizes three essential steps of dynamic analysis to transform source specifications into a trace tree representation. These sub-steps are instrumentation, execution and visualization. Instrumentation step is responsible to instrument the source specifications by automatically inserting probes inside the source code. Execution step takes the instrumented specifications along with real input data and runs the instrumented specifications. Each probe activation is recorded during the Execution step. Visualization step displays the results in the form of a trace tree and bounds original source specifications to the corresponding trace tree representation.

4.1.1 Instrumentation of Source Specifications

Input to the instrumentation step is original specifications and the output is the instrumented specifications. Instrumentation is performed by automatically inserting probes inside the source specifications. A probe is a statement added to a statement of the original source code, with no disruption to its semantics. The probe is written to extract the required information during execution. The base for the code modification is the Abstract Syntax Tree (AST) produced by the Smalltalk environment. Instrumentation step generates probing messages in four sub-steps as shown in Fig. 3.

The first sub-step is to parse the source code for AST generation. The second sub-step is to instrument the AST. This sub-step automatically generates additional nodes in the original AST. The output is an instrumented AST. The third sub-step is to compile the instrumented AST. The output is the compiled source code. Finally, the original source code is replaced with the compiled source code.

4.1.2 Execution of Instrumented Specifications

It performs trace recording through the activation of probes in the instrumented code. The input to this step is the instrumented source code in the form of probing messages from the instrumentation phase. In addition to the instrumented code, real input data is provided to execute the instrumented code.

Once the instrumented code and the input data are available, the trace recording is done through the activation of probes in the instrumented code and the recording of events in the trace. Each probe is implemented as a message sent to a collector along with the information from the execution context. The collector receives the message, creates a corresponding record event and adds it to the trace tree.

4.1.3 Trace Visualization

It binds each event in the trace to the original source code in the form a trace tree as shown in Fig. 4. The right hand side represents the original source code while the left hand side represents its trace tree representation. Each entity in the source code, such as different variables and operators on the right hand side, is linked to the corresponding trace tree entity. It is illustrated by drawing arrows in Fig. 4.

It means that one can go through all the application source code, starting from the beginning and observe the corresponding arrangement in the trace tree for each single element. It may help in comprehension of the source code.

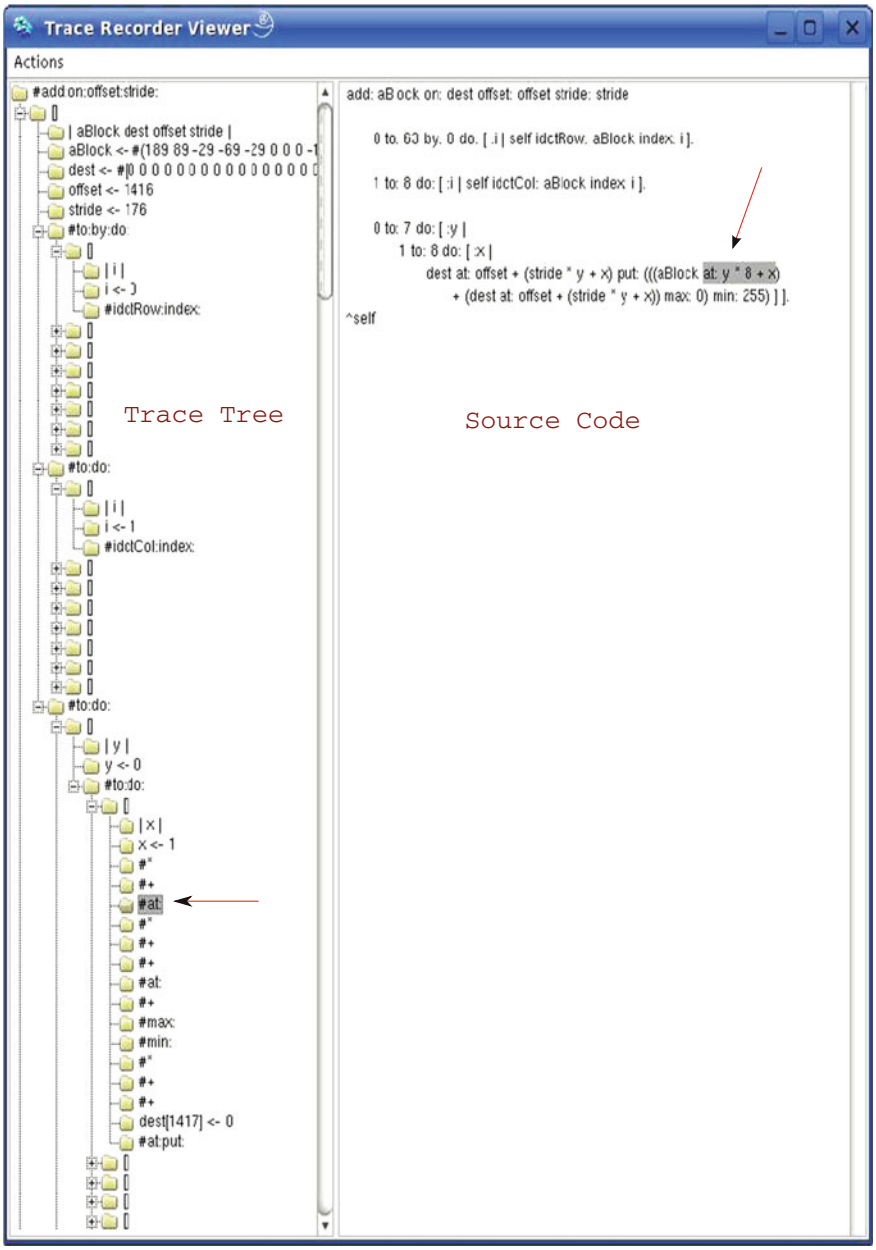


Fig. 4 Trace tree representation and corresponding source specification

4.2 Trace Tree Analysis

The output of first part is a trace tree which represents the sequence of recorded events in a tree-like form. A typical use of the trace tree is to hierarchically show the structure of function calls during a particular execution of the source specification. Once source specification is transformed into a trace tree representation, we perform operations on the trace tree for different types of analysis.

4.2.1 Multiple Analysis Operations on Trace Tree

Multiple analysis operations can be performed on basic entities of the trace tree. For example:

- Checking the value of each variable in every step of the program execution
- In the context of code rewriting, one may perform operations for type-checking, code optimization and flow analysis
- For early design space exploration, application orientation and extraction of spatial parallelism are the key concerns.

4.2.2 Different Visitors for Multiple Analysis Operations

We keep basic entities of the trace tree independent from analysis operations that apply to them. Related operations that we want to perform on the trace tree entities are packaged in a separate object named as "visitor" and pass it to the elements of trace tree. There are different visitors for multiple analysis operations. We have already explained the concept of visitor design patten in Sect. 3.

The proposed analysis framework is generic as it is not restricted to a particular set of analysis operations. It allows the designer to extend the framework by defining new analysis operations to fulfill different requirements of design space exploration. For each analysis operation, there is a corresponding visitor for the trace tree, making a visitor hierarchy similar to the visitors on a parse tree [34]. For example, in this article, we have defined visitors for application orientation and spatial parallelism information. However, the framework can easily be extended by simply defining new visitors.

5 Usage Scenarios in a Holistic Design Environment

Section 4 presented a generic application analysis framework for application characterization at higher abstraction level. The proposed analysis framework can perform multiple analysis operations on the trace tree representation of source specifications, depending upon a particular requirement of the design space exploration. This section provides some usage scenarios of the proposed framework in a holistic design

environment. It includes: (1) application orientation, (2) spatial parallelism, (3) guidelines for mapping and (4) guidelines for performance estimation.

5.1 Application Orientation

Application orientation provides guidelines about architecture selection. An application may have three types of operations: computational operations, memory operation and control operations. Our analysis results describe the application in terms of percentages of these three basic types of operations.

Computation Oriented Operations include arithmetic operations such as addition, multiplication, subtraction etc. and logical operations such as "and", "or" etc. The analysis results show the percentage of each type of computation operations in a function. For example, if most of the operations are multiplications, then target architecture should have dedicated hardware multipliers, hence guiding the designer towards architecture selection.

Memory Oriented Operations tell the designer that a particular function is data access dominated and is most likely to require a high data bandwidth. It indicates that the computations are not performed on previously computed data reside in local memories but performed on the input data entering into the trace tree. Therefore, in the case of real time constraints, some efficient mechanism of data movement and high performance memories are required.

Control Oriented Operations guide the designer to evaluate the need for complex control structures to implement a function. The functions with high percentage of this types of operations are good candidate for a GPP implementation rather than a DSP implementation, since the latter is not well suited for control dominated functions. In addition to this, a hardware implementation of these control dominated functions would require large state machines.

5.2 Spatial Parallelism

Trace tree representation shows the existing parallelism among the operations of the function. It implies the possibility of mapping different operations or functions to different processing elements of the target architecture for concurrent execution. In other words, we can exploit the inherited spatial parallelism present in the application.

We represent the amount of average inherited spatial parallelism for every function in the source specification by *"P"* such that functions with higher *"P"* values are considered as appropriate to architectures with large explicit parallelisms. Functions with lower *"P"* value are rather sequential and acceleration can only be obtained by exploiting temporal parallelism.

Definition 7.4 *Average Inherited Spatial Parallelism (P)*—The value of average inherited spatial parallelism at any hierarchical level of a trace tree is computed by dividing the total number of operations by its critical path.

Definition 7.5 *Critical Path*—The critical path at any hierarchical level of a trace tree is the number of longest sequential chain of operations (processing, control, memory). It is computed for each hierarchical level.

When we compute the value of *"P"* for any hierarchical level in a trace tree, we assume that the parallel execution of sub-hierarchical levels is possible and the value of *"P"* is given as the ratio between the sum of all operations in the sub hierarchical levels of the node and the longest of all the critical paths.

For example, if a node "A" in the trace tree has three sequential sub-nodes "B", "C" and "D" containing 10, 20 and 30 sequential operations respectively. Now the value of "P" at each sub-node "B", "C" and "D" is 1 as they contain only sequential operations and hence no spatial parallelism. However, the value of "P" at node "A" is 2 (60 divided by 30) assuming that all the sub-nodes can be executed in parallel.

5.3 Guidelines for Mapping

The mapping process requires application model in the form of different functions as well as architecture model in the form of processing elements and interconnections to map the application behavior on the architecture model. The obtained analysis results identify the most complex functions in terms of computations, which may be the best candidates for mapping to the fastest processing elements (PEs). The designers also prefer to map the functions which communicate heavily with each other to the same PE or to the PEs connected by dedicated busses.

5.4 Guidelines for Performance Estimation

The performance estimation of different functions of the application on multiple processing elements of the architecture is another important issue in the design space exploration. For example, assuming a function F1 is mapped to processing element PE1. If F1 contains "X" integer-type multiplication operations and executing such an operation on PE1 requires "Y" clock cycles (known to designer from architecture model). Then the execution time of function F1 on processing element PE1 will be: $\textit{Execution Time} = (X) * (Y) = XY \textit{ clock cycles}$.

6 Experimental Results

The purpose of this section is to provide analysis results for MPEG-2 video decoding application [20]. The basic principles of MPEG-2 decoding application are first described in Sect. 6.1. A Smalltalk implementation of MPEG-2 decoder is used to perform experiments in Sect. 6.2.

6.1 MPEG-2 Video Decoding Application

MPEG-2 is a generic coding method of moving pictures and of associated audio because it serves a wide range of applications, bit-rates and resolutions [20]. The basic principle is to remove the redundant information prior to transformation and re-inserting it at the decoder. There are two types of redundancies: spatial redundancy and temporal redundancy. Spatial redundancy is the correlation of pixels with their neighbouring pixels with in the same frame. Temporal redundancy is used to remove the correlation of pixels with neighbouring pixels across the frames.

MPEG-2 video decoding process is shown in Fig. 5. Input to the decoder is the incoming MPEG-2 video bit-stream. The first step is to perform Huffman decoding which generates: (1) quantized macroblocks encoded in the frequency domain (2) predictively encoded motion vectors. In the subsequent steps, the quantized macroblocks are inverse transformed while the motion compensation is performed to decode offset encoded motion vectors.

Inverse transformation is due to spatial redundancy reduction at encoder. It maps each 8×8 block from the frequency domain back to the spatial domain. Each block is reordered, inversely quantized and then followed by an two-dimensional (2D) Inverse Discrete Cosine Transform (IDCT). Similarly, encoded motion vectors are decoded. Motion compensation is due to the temporal redundancy reduction at the encoder side and recovers predictively coded macroblocks. It uses motion vectors to find a corresponding macroblock in a previously decoded reference picture. Frame

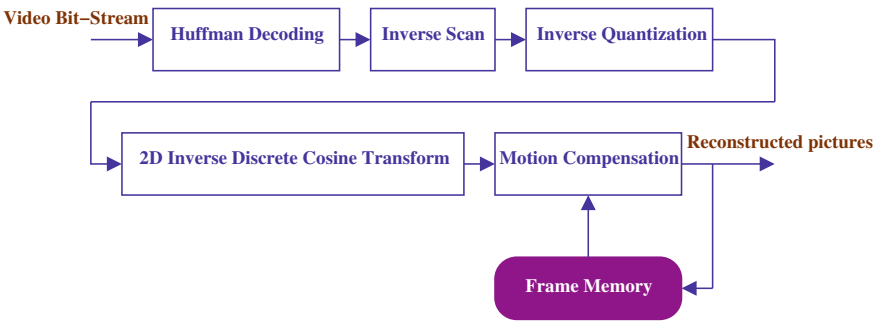


Fig. 5 MPEG-2 video decoding process

Table 2 Orientation results for 2D-IDCT

Function	Computation	Memory	Control	P
idctCol:index:	76.36	23.64	0	1
idctRow:index:	76.36	23.64	0	1
add:on:offset:stride:x	77.11	22.89	0	24.14

memory is used to store the reference frames. The reference macroblock is added to the current macroblock, to recover the original picture data.

6.2 Application Orientation and Spatial Parallelism Results

This section performs experiments with different blocks of MPEG-2 video decoder to get analysis results in terms of application orientation and spatial parallelism. For simplicity, we present experimental results with Two Dimensional Inverse Discrete Cosine Transform (2D-IDCT) and Huffman Decoding to illustrate our analysis approach.

6.2.1 Inverse Discrete Cosine Transform

2D-IDCT for 8×8 image blocks is transformed into a trace tree using the flow in Fig. 3. We perform analysis operations on the trace tree representation to get analysis results. Table 2 shows the analysis results for different functions in 2D-IDCT. The first column represents the name of method. The second, third and fourth columns represent the percentages of computation, memory and control in each method respectively. The last column represents the amount of inherited spatial parallelism.

From the structural point of view, 2D-IDCT is composed of two identical and sequential one-dimensional IDCT (1D-IDCT) sub-blocks, operating on rows and columns. The method *"idctCol:index:"* and method *"idctRow:index:"* in Table 2 represent 1D-IDCT operations on columns and rows respectively. The corresponding trace trees have the same orientation values for both methods as shown in Table 2. The method *"add:on:offset:stride:"* in Table 2 represent 2D-IDCT.

The first observation is that the percentage of control operations is zero since it is composed of deterministic loops and does not contain any test. Secondly, the computation percentage for 2D-IDCT functional blocks are higher so it is computation oriented. The results also show a good percentage of memory operations.

Figure 6 shows the percentage of each type of computation in 2D-IDCT. It does not contain any floating point operations. It implies that processors with dedicated floating point units are not necessary and processor selection should focus on integer performance. Furthermore, 27 % operations are multiplications such that selected processors may have dedicated hardware multipliers.

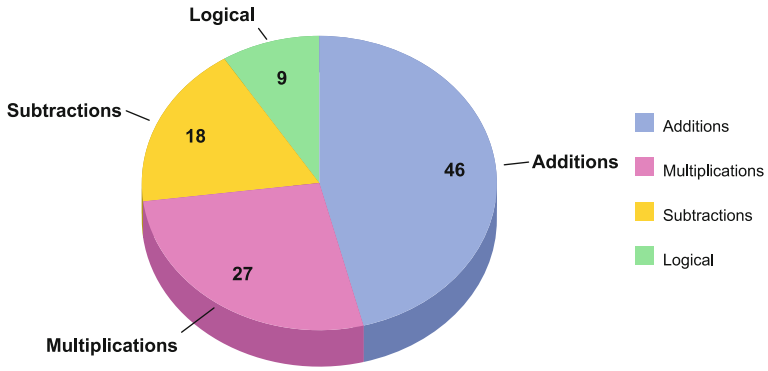


Fig. 6 Percentages of computation types for 2D-IDCT

In Table 2, the lowest level of granularity is exhibited by 1D-IDCT sub-blocks operating on rows and columns. The value of "P" is 1 indicating no fine grain spatial parallelism. It shows that these sub-block methods are sequential in nature and do not contain any inherited spatial parallelism. However, the amount of spatial parallelism increases at the higher level of granularity. The value of "P" at this level is 24.14, indicating that a coarse grain parallelism is available.

The fact that there is no need for complex control structures, the high data-accesses and the coarse grain parallelism implies that optimizations can be obtained with a pipelined architecture with possible coarse grain dedicated hardware modules providing a large bandwidth. So if high performances are required, an ASIP (Application Specific Instruction-set Processor) or a programmable dedicated hardware can be introduced within the SoC.

6.2.2 Huffman Decoding

Table 3 shows the analysis results for huffman decoding methods. It can be noticed that these functions have relatively high percentages of control operations denoting heavily conditioned data-flows. The percentage of computation operations also indicates an important computation frequency. There are less number of memory operations as compared to computations and control operations. It indicates that these methods are control and computation oriented.

Figure 7 shows the orientation of Huffman Decoding. There are no multiplications, so selected processors have no need for dedicated hardware multipliers. The results show that 45 % of the computations are logical operations.

We have not shown the value of P in Table 3 because the value of P remains 1 (the value of P for sequential code) at all hierarchical levels of the trace tree. It reveals that suitable target architecture for Huffman decoding algorithm may be a GPP. There is no need for a DSP and for a complex data path structure, since there is no spatial parallelism at any hierarchical level.

Table 3 Orientation results for huffman decoding

Function	Computation	Memory	Control
getChromaDCDctDiff	49	2	49
getCodedBlockPattern	52	5	43
getLumaDCDctDiff	60	2	38
getMacroblockAddrIncrement	50	5	45
getMacroblockMode:	58.3	8.4	33.3
getMotionDelta:	58.2	3	38.8
getQuantizerScale:	75	0	25
Huffman Decoding	60	7	33

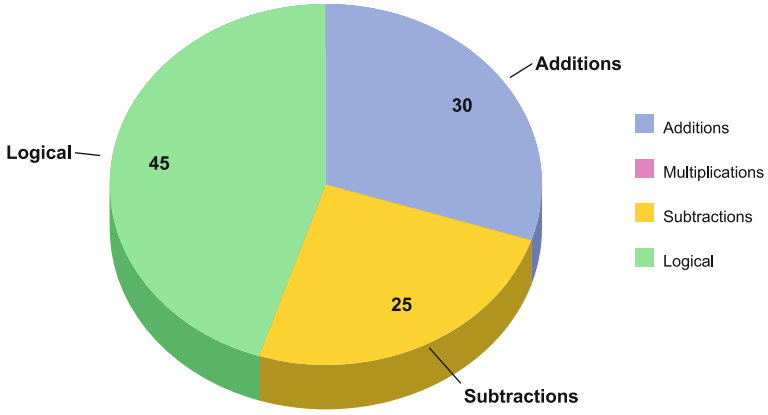


Fig. 7 Percentages of computation types for huffman decoding

This section has presented the analysis results for 2D-IDCT and Huffman Decoding in MPEG-2 decoder in terms of application orientation and spatial parallelism. The next section will further illustrate the significance of spatial parallelism information by representing the application in the form of parallel process networks.

7 Process Oriented Application Descriptions

We have explained in the introductory part of this chapter that the performance gain in multi-core architectures depends upon application parallelization across the cores. It requires the extraction of inherited spatial parallelism present in sequential applications written in high level languages (such as C, C++ Smalltalk). Section 6 provided spatial parallelism results for MPEG-2 video decoding application. In this section, we exploit the spatial parallelism information and represent the application in the form of parallel process networks by using AVEL framework. The objective is to describe source specification of the application in form of parallel process networks to perform parallel and distributed computations. The state of the art in parallel process networks representation is described in Sect. 2.3.

7.1 Syntax of AVEL Framework

AVEL framework specifies three kinds of processes which are composed hierarchically. The first type is the Primitive Process which is the leaf of a process network hierarchy and implements an atomic behavior. The second type is the Node Process which is the composition of other processes and behaviors. It allows an hierarchical description of process network. The third type is the Alias Process which is declared outside the main process and is re-used by another processes. We use Alias Process to factorize complex behaviors in the code. The syntax to declare the Primitive Process or the Node Process is given as:

Process Name {Output Connections} [Behavior]

The Process Name is used as an identifier in the process network. To simplify connections between different nodes of the process network, only the output connections are declared. The Behavior of a process can be atomic or composite. For the Primitive Process, the atomic behavior is an identifier used to make a link with its corresponding function in the smalltalk specification. For the Node Process, the composite behavior corresponds to a sub-network of processes such that the first process is connected to the input ports of its hierarchy and the last process is connected to the output ports. For example, if the output of the process "ProcessA" with behavior "BehaviorA" is connected to the process "ProcessB" at output port "1", then it is specified as:

ProcessA {ProcessB@1} [BehaviorA]

The graphical representation of the Primitive Process and the Node Process syntax is shown in Figs. 8 and 9 respectively.

The syntax to declare an Alias Process is given as:

Process Name (process name) [Output Connections]

The graphical representation of the Alias Process syntax is shown in Fig. 10.

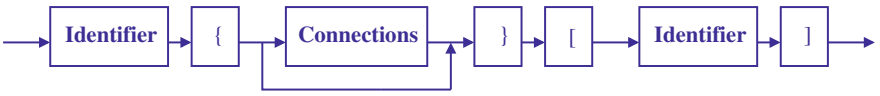


Fig. 8 Primitive process syntax (represents a leaf in process network hierarchy)

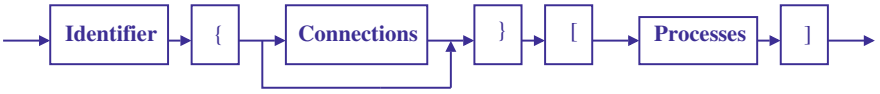


Fig. 9 Node process syntax (represents hierarchical description of a process network)

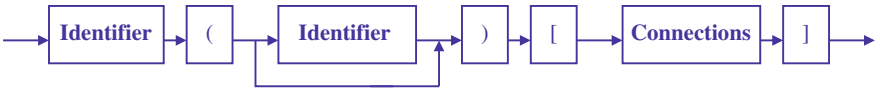


Fig. 10 Alias process syntax (factorizes complex behaviors in the program)

7.2 Example of AVEL Program

In order to illustrate the syntax of AVEL processes, an AVEL program "Example" with hierarchical composition of the processes is shown below. The graphical representation of the "Example" program is shown in Fig. 11.

```
01.  StrZ{}
02.  [
03.       Prim1{Prim2@1}[Prim1]
04.       Prim2{}[Prim2]
05.  ]

06.  Example{}
07.  [
08.       Split{StrA@1 StrB@1} [splitter]
09.       StrA{StrZ}{Join@1}
10.       StrB{StrC@1}
11.       [
12.            PrimA {PrimB@1}[prima]
13.            PrimB{}[primb]
14.       ]
```

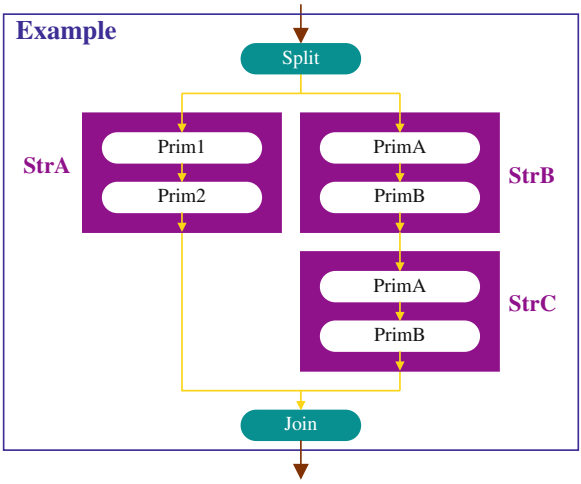


Fig. 11 Graphical representation of "Example" program (takes an input stream and splits it into two processes

```
15.       StrC{StrB}{Join@2 }
16.       Join{}[joiner]
17.   ]
```

"Example" (line 6) is an AVEL process network that takes an input stream and splits it into two other processes *"StrA"* and *"StrB"* (line 8). *"StrZ"* (line 01) is a Node Process because its behavior contains other Primitive Processes *"Prim1"* and *"Prim2"* (line 03 and 04 respectively). *"StrB"* (line 10) is also a Node Process because its behavior contains other Primitive Processes *"PrimA"* and *"PrimB"* (line 12 and 13 respectively). *"StrA"* (line 09) and *"StrC"* (line 15) are Alias Processes because their behaviors reuse predefined processes *"StrZ"* and *"StrB"* respectively. "Split" (line 08 in above program) and "Join" (line 16 in above program) are two Primitive Processes: Former is responsible for distributing input stream between its outputs while the latter is responsible for merging an output stream from its inputs.

7.3 MPEG-2 Video Decoder in AVEL

We have described MPEG-2 video decoding in Sect. 6.1. However, it did not explain the parsing of incoming video bit stream into an object stream. As its name implies, parser reads the incoming video bit stream to extract different syntactic structures. The process of parsing the incoming video bit-stream consists of many layers of nested control flow. It makes the parser unsuitable for streaming computations. As AVEL is intended for streaming computations, parsing of MPEG-2 bit stream into an object stream is implemented in a higher level language like Smalltalk rather than AVEL.

The transformation of video bit-stream into an object stream ensures that all syntactic structures above macroblocks have been treated. The following AVEL program shows slice (a collection of macroblocks) processing in MPEG-2 decoder. Figure 12 shows the graphical representation of the *"ProcessSlice"* AVEL Program.

```
01. ProcessSlice {} [
02.       MBAddr{MBMode@1}[mbaddr]
03.       MBMode {Split@1}[mbmode]
04.       Split {IntraMB@1
05.              FieldMB@1
06.              FrameMB@1
07.              Pattern@1
08.              DMV@1
09.              NoMotion@1}[splitter]
10. IntraMB {join@1}
11. [
12.       VLD{InverseScan@1}[vld]
13.       InverseScan{InverseQuant@1}[is]
```

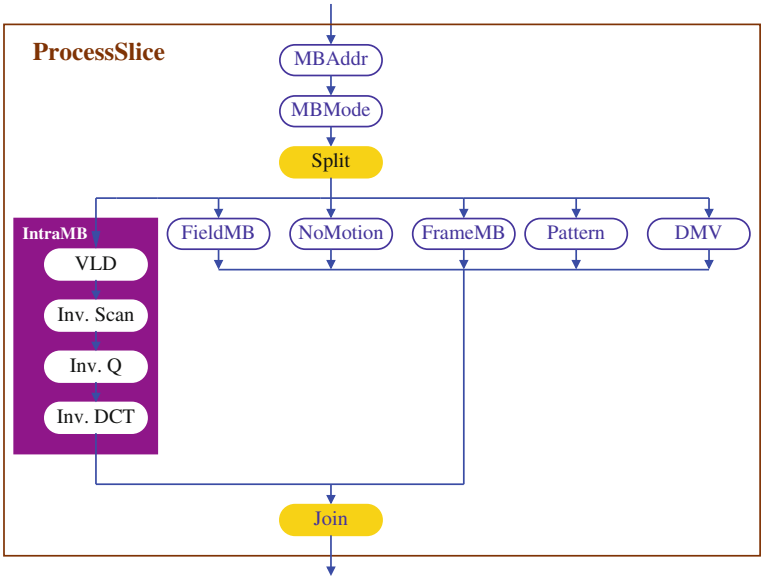


Fig. 12 Graphical representation of ProcessSlice AVEL program

```
14.       InverseQ{IDCT@1}[iq]
15.       IDCT {}[idct]
16. ]
17. FieldMB {} {Join@2}
18. FrameMB {} {Join@3}
19. Pattern {} {Join@4}
20. DMV {} {Join@5}
21. NoMotion{} {Join@6}
22. Join {} [joiner]
23. ]
```

The slice processing (line 01 in the above program) starts by calculating the macroblock address increment (line 02 in the above program and shown as *"MBAddr"* in Fig. 12). It indicates the difference between current macroblock address and previous macroblock address. We have implemented it as a primitive process with behavior *"mbaddr"*. The behavior of this process contains inherent parallelism. We can implement this process as node process which contains other primitive processes. However, we have shown it as a primitive process in Fig. 12) for simplicity. After calculating macroblock address increment, macroblock mode (line 03 in the above program) is calculated which indicates the method of coding and contents of the macroblocks according to tables defined in MPEG-2 standard [20]. Each type of macroblock is treated differently. We have implemented it as a primitive process with behavior *"mbmode"*. However, we can implement it as node process containing other primitive processes.

The output of *"MBMode"* is given to any of the six processes (line 04–09 in the above program). All of these processes *"IntraMB"*, *"FieldMB"*, *"FrameMB"*, *"Pattern"*, *"DMV"* and *"NoMotion"* are node processes and consists of other primitive processes. But for simplicity, we have shown only *"IntraMB"* as node process (line 10 in the above program) and all of other processes are shown as primitive processes.

"IntraMB" further consists of primitive processes. These processes are *"VLD"*, *"InverseScan"*, *"InverseQ"*, and *"IDCT"* (line 12–5 in above program). Again, we have implemented all of these processes as primitive process. We can implement these processes as node process which contains other primitive processes depending upon the amount of spatial parallelism obtained from analysis framework.

8 Conclusions

In this article, we proposed a generic application analysis methodology for early design space exploration of embedded systems. Source specifications in a high level object oriented language Smalltalk were transformed into a trace tree representation by dynamic analysis. Unlike conventional dynamic analysis techniques, code instrumentation was performed on abstract syntax tree rather than source code. We executed the instrumented application to get trace tree representation. A generic application analysis approach was used to characterize the trace tree representation. Unlike conventional analysis approaches, the proposed approach was not restricted to a set of particular design metrics.

To illustrate the significance of proposed framework, we performed analysis operations on the trace tree representation of MPEG-2 video decoding application and obtained application characterization results in terms of: (a) processing, control and memory orientations (b) spatial parallelism. We used spatial parallelism information to model computational intensive part of the source specifications in the form of parallel process networks.

References

1. S.S. Bhattacharyya et al., Overview of the MPEG reconfigurable video coding framework. J. Signal Process. Syst. **63**(2), 251–263 (2011)
2. Y.Q. Shi, H. Sun, *Image and video compression for multimedia engineering: fundamentals, algorithms, and standards*, 2nd edn. (Cambridge, Massachusetts, USA, 2008)
3. G. Ascia, V. Catania, A.G. Di Nuovo, M. Palesi, D. Patti, Efficient design space exploration for application specific systems-on-a-chip. J. Syst. Arch. **53**(10), 733–750 (2007). Special Issue on Architectures, Modeling, and Simulation for Embedded Processors
4. I. Bacivarov, W. Haid, K. Huang, L. Thiele, Methods and tools for mapping process networks onto multi-processor systems-on-chip, in *Handbook of Signal Processing Systems*, Part 4 (2010), pp. 1007–1040

5. J.P. Diguet, Y. Eustache, G. Gogniat, Closed-loop based self-adaptive HW/SW embedded systems: design methodology and smart cam case study. ACM Trans. Embed. Comput. Syst. **10**(3) (2011)
6. S. Abdi, Y. Hwang, L. Yu, G. Schirner, D. Gajski, Automatic TLM generation for early validation of multicore systems. IEEE Des. Test Comput. **28**(3), 10–19, May–June 2011
7. C. Silvano, et al., Parallel programming and run-time resource management framework for many-core platforms. In 6th International Workshop on Communication-centric Systems-on-Chip, Aug 2011, pp. 1–7
8. S. Rul, H. Vandierendonck, K.D. Bosschere, Function level parallelism driven by data dependencies. SIGARCH Comput. Archit. News **35**(1), 55–62 (2007)
9. J. Cockx, K. Denolf, B. Vanhoof, R. Stahl, SPRINT: A tool to generate concurrent transaction-level models from sequential code. EURASIP J. Adv. Signal Process. **2007**, article ID. 75373, 15 pp. (2007)
10. W. Thies, V. Chandrasekhar, S. Amarasinghe, A practical approach to exploiting coarse-grained pipeline parallelism in C programs, in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-40)*, Illinois, USA, Chicago, Dec 2007, pp. 356–369
11. W. Thies, S. Amarasinghe, *An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design*, PACT Austria, Sept 2010, pp. 11–15
12. N.B. Amor, Y.L. Moullec, J.P. Diguet, J.L. Philippe, M. Abid, Design of a multimedia processor based on metrics computation. Adv. Eng. Soft. **36**(7), 448–458 (2005)
13. Y. Moullec, J.P. Diguet, N. Amor, T. Gourdeaux, J.L. Philippe, Algorithmic-level specification and characterization of embedded multimedia applications with design trotter. J VLSI Signal Process. Syst. **42**(2), 185–208 (Feb 2006)
14. A. Prihozhy, M. Mattavelli, D. Mlynek, Evaluation of the parallelization potential for efficient multimedia implementations: dynamic evaluation of algorithm critical path. IEEE Trans. Circuits Syst. Video Technol. **93**(8), 593–608 (2005)
15. C. Lucarz, G. Roquier, M. Mattavelli, High level design space exploration of RVC codec specifications for multi-core heterogeneous platforms, in *DASIP* (2010), pp. 191–198
16. I. Tomek, The joy of smalltalk, Feb 2002, 700 pp [Online]. Available:www.iam.unibe.ch/ducasse/FreeBooks/Joy/
17. B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke, A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Soft. Eng. **35**(5), (2009)
18. A. Sengupta, R. Sedaghat, Z. Zeng, Multi-objective efficient design space exploration and architectural synthesis of an application specific processor (ASP). J. Microprocessors Microsyst: Embed. Hardware Des. (MICPRO) **35**(4), (2011)
19. W. Thies, An empirical characterization of stream programs and its implications for language and compiler design, in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10*, Vienna, Austria, Sept 2010
20. ISO/IEC 13818–2, Generic coding of moving pictures and associated audio information-part 2: video, 1994, also ITU-T Recommendation H.262
21. H. Koziolek, Performance evaluation of component-based software systems: A survey.J. Perform. Eval. **67**(8) (Elsevier Science, 2010)
22. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom, The worst-case execution-time problem: overview of methods and survey of tools. Trans. Embed. Comput. Syst. **7**(3) (2008) article 36
23. M. Ravasi, M. Mattavelli, High abstraction level complexity analysis and memory architecture simulations of multimedia algorithms. IEEE Trans. Circuits Syst. Video Technol. **15**(5), 673–684 (2005)
24. V. Uquillas Gomeza, S. Ducasse, T. D'Hondta, Ring: A unifying meta-model and infrastructure for smalltalk source code analysis tools. Comput. Lang. Syst. Struct. **38**(1), 44–60 (2012)
25. R. Stephens, A survey of stream processing. Acta Informatica **34**(7), 491–541 (July 1997)
26. W.R. Mark, R.S. Glanvilley, K. Akeleyy, M.J. Kilgardy, Cg: A system for programming graphics hardware in a C-like language. ACM Trans. Graph. **22**(3), 896–907 (2003)

27. N. Goodnight, R. Wang, G. Humphreys, Computation on programmable graphics hardware. IEEE Trans. Compu. Graph. Appl. **25**(5), 12–15 (2005)
28. S. Yamagiwa, L. Sousa, Modeling and programming stream-based distributed computing based on the meta-pipeline approach. Int. J. Parallel Emergent Distrib. Syst. **24**(4), 311–330 (2009)
29. C. Consel, H. Hamdi, L. Reveillere, L. Singaravelu, H. Yu, C. Pu, Spidle: A DSL approach to specifying streaming applications, in *Proceedings of 2nd International Conference on Generative Programming and Component, Engineering (GPCE'03)* (2003), pp. 1–17
30. M. Gokhale, J. Stone, J. Arnold, M. Kalinowski, Stream-oriented FPGA computing in the streams-C high level language, in *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*, Napa Valley, California, USA, Apr 2000, pp. 49–56
31. J.L. Gaudiot, T. DeBoni, J. Feo, W. Bohm, W. Najjar, P. Miller, The Sisal Model of functional programming and its implementation, in *Proceedings of the 2nd International Aizu Symposium on Parallel Algorithms/Architecture Synthesis*, Aizu-Wakamatsu, Fukushima, Japan, March 1997, pp. 112–123
32. S. Malek, N. Esfahani, D.A. Menasc, J.P. Sousa, H. Gomaa, Self-architecting software systems (SASSY) from QoS-annotated models, in *Proceedings of the Workshop on Principles of Engineering Service Oriented Systems*, Canada, 2009
33. C.-K. Lin, A.P. Black, DirectFlow: A Domain-specific language for information-flow systems. Lect. Notes Comput. Sci. **4609/2007** 299–322 (2007)
34. E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series, 1995)

Part IV
Performance Analysis, Power Management and Deployment

An Efficient Cycle Accurate Performance Estimation Model for Hardware Software Co-Design

Muhammad Rashid

Abstract Software performance in terms of clock cycles can be measured on the hardware platform. However, the availability of hardware platform is critical in early stages of the design flow. One possible solution is to implement the hardware components at cycle-accurate level such that the performance estimation is given by the micro-architectural simulation in number of cycles. But the design space exploration at this level may require huge simulation time. This article presents a cycle-accurate performance estimation methodology with reduced simulation time. The simulation results are computed and stored in a performance estimation database. The results are used for mapping application functions on architecture components. We estimate the application performance as a linear combination of function performances on mapped components. The proposed approach decreases the overall simulation time while maintaining the accuracy in terms of clock cycles. We have evaluated the design with H.264 application and found that it reduces 50 % of the simulation time.

1 Introduction

Software is a dominant part of current embedded applications due to flexibility, time-to-market and cost requirements issues. In HW/SW co-design process, software development is responsible for more than 75 % of the design delays [7]. This delay exists because the system is poorly conceived and complex algorithms fail to adequately address systems performance. Comparison of estimated application behavior on a number of different hardware components is called performance estimation. A key to successfully accomplishing this task is the design space exploration (DSE). It involves simulation of different implementation alternatives for software performance estimation.

M. Rashid (✉)
Umm Al-Qura University, Makkah, Saudi Arabia
e-mail: mfelahi@uqu.edu.sa

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_8,

Software performance estimation of a function on each processing element is a very time consuming and difficult task. Any change in the hardware architecture may require a new partitioning scheme for the associated software. There are three requirements on the software performance estimation technique. First, it should be adaptable to reflect varying architecture features. Secondly, it should take into account the compiler options. Finally, the performance estimation technique should consider the data-dependent performance variations. Therefore, fast software performance estimation tools, to perform simulations at different abstraction levels, are needed in early stages of the design flow [12].

The well-known abstraction levels for performing simulations are Register Transfer Level (RTL), Transaction Level Modelling (TLM) and cycle-accurate. Performance estimation at RTL requires great quantities of simulation time to explore the huge architectural solution space [19]. TLM tools reduces simulation time compared with RTL by using higher abstraction of inter component communication activity with acceptable cost of timing accuracy. The problem with TLM based techniques is the lack standardization. Consequently, industry/research labs have developed their own internal TLM standards such that the models from different providers have different degree of accuracy [1]. Cycle-accurate simulations [3] is another alternative for performance estimation. Despite of accurate performance estimation, it is often too slow to be used inside the DSE loop.

This article proposes a DSE framework, which consists of five stages. However, the core of the framework is the second stage in which a software performance estimation methodology at cycle-accurate level is presented. Application behavior is modeled as a composition of function blocks. The performance of each function block on different processing elements is stored in a performance estimation database. The information in the database is used for mapping different function blocks in application software to different processing elements in hardware architecture. Once mapping decision is made, we estimate the system performance as a linear combination of function block performances on mapped components.

The focus of performance estimation technique in this article is on simulation of individual processing elements (PEs), which forms a major bottleneck in achieving high simulation speeds. Traditionally, individual PEs have been simulated using Instruction Set Simulators (ISS). A number of recent works have suggested various ISS acceleration techniques, such as compiled simulation [17] or just-in-time compiled simulation [2]. However, due to the increasing complexity of MPSoCs, even such improvements are not enough to achieve the desired simulation speed. We will describe the ISS acceleration techniques in Sect. 2 of this article.

In order to implement the proposed performance estimation methodology, a simulation platform is required which: (1) models all architecture features and (2) performance estimation is made with the binary executable after compilation. We use SoCLib library [21], an open source platform for virtual prototyping of MPSoCs, to describe architectural components.

We build a simulation platform by instantiating different modules from the SoCLib library and execute the considered application on the target simulation platform. The core of the platform is a library of SystemC simulation moules. However, the

native SoCLib simulation modules are not sufficient to implement the proposed performance estimation methodology. Consequently, we create one new module and modify one of the native module in the SoCLib library.

The SoCLib simulator models all architecture features and estimation is made with the binary executable after compilation. Therefore, first two requirements of the performance estimation, consideration of architectural features and compiler optimizations, are satisfied. The third requirement, consideration of data dependent behavior, is met by simulating each function with different input data. Accordingly, we compute the Worst Case Execution Time (WCET) and the Average Case Execution Time (ACET) for individual functions blocks.

To summarize, the contributions of this article are as follows:

- A complete DSE framework is proposed. It consists of five stages. The in-depth description of the second stage, a performance estimation methodology at cycle-accurate level with reduced simulation time, is presented.
- In order to implement the proposed cycle-accurate performance estimation methodology, SoCLib simulation library is extended.

The rest of this article is organized as follows: Sect. 2 describes the related work in software performance estimation. Section 3 presents a DSE framework with five stages. Section 4 presents the performance estimation stage of the proposed framework. Simulations are performed and the performance results of individual function blocks are stored in a performance estimation database. Section 5 presents SoCLib library of simulation models. Experimental results with H.264 video encoding application are provided in Sect. 6. Finally, we conclude the article in Sect. 7.

2 Review of Related Work

This section presents various simulation techniques for performance estimation proposed in recent works. We divide the existing simulation techniques into four categories as shown in Fig. 1. The four simulation techniques are: instruction set simulation, partial simulation, annotation-based simulation and hybrid simulation.

2.1 Instruction Set Simulation

An Instruction Set Simulator (ISS) functionally mimics the behavior of a target processor on a host work station. It is further subdivided into interpretive simulation and compiled simulation.

Interpretive Simulator is a virtual machine implemented in software. An instruction word is fetched, decoded and executed at runtime in a simulation loop. SimpleScalar [5] and SimOS [18] are the typical examples. Interpretive Simulation is the basic ISS technique which is flexible but slow. To increase the simulation speed, the concept of compiled simulation was proposed.

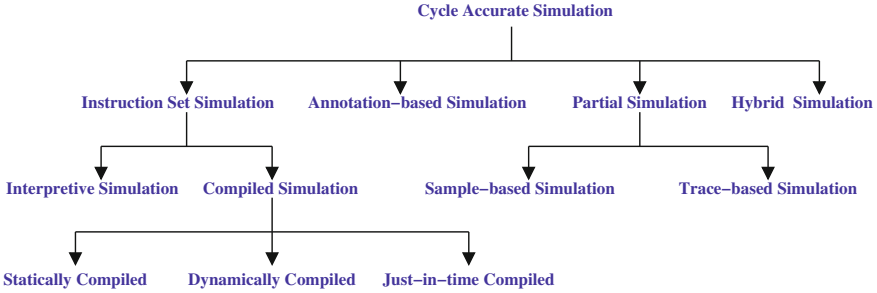


Fig. 1 Classification of simulation techniques

Compiled Simulation is used to improve the simulation performance by shifting the time consuming operations, such as instruction fetching and decoding, from runtime to compile time. It is further subdivided into dynamically compiled, statically compiled and Just-in-time (JIT) compiled simulations. Dynamically compiled and statically compiled techniques have in common that a given application is decoded at compile time. In order to compare dynamic and static compilation with interpretive simulation, we divide the processing of application code into three steps: instruction decoding, operation sequencing and operation scheduling as shown in Table 1.

In interpretive simulation technique, all processing phases are performed at runtime [5, 18]. In dynamically compiled simulation technique, instruction decoding and operation sequencing phases are performed at compile time, while the operation scheduling phase is still performed at runtime [17]. In statically compiled simulation technique, all the three phases are performed at compiled time [4]. A JIT compiled simulation technique [2] exploits special features of architecture description languages to combine flexibility and high simulation speed. The compilation of target binary takes place at runtime and the result is cached for reuse.

2.2 Partial Simulation

Partial simulation techniques are used to obtain performance estimation of the whole application without having to simulate it to completion. It is further subdivided into sampling-based and trace-based partial simulation techniques.

Table 1 Application code processing phases in different simulation techniques

Simulation technique	Instruction decoding	Operation sequencing	Operation scheduling
Interpretive simulation [5, 18]	Runtime	Runtime	Runtime
Dynamically compiled [17]	Compile time	Compile time	Runtime
Statically compiled [4]	Compile time	Compile time	Compile time

Sampling-based Partial Simulation Technique advocates the sampling of selected phases of an application to estimate the performance of a processing element. An analytical sampling technique is presented in [20]. Representative samples are selected by analyzing the similarity of execution traces. Samples are represented by basic block vectors and can be obtained by performing a functional simulation at the pre-processing phase. SMARTS [22] is another sampling microarchitecture simulator. Functional simulation can be used to fast-forward the execution until samples are met. Detailed simulation is performed on these samples, and the obtained performance information is used to extrapolate that of the whole application.

Trace-based Partial Simulation Technique generates a synthetic trace to represent the performance. A trace is some information of interest generated during the execution of a program on a simple and fast simulation model. Later, analysis tools can process the traces off-line in a detailed fashion. An example of trace-based partial simulation is [6].

2.3 Annotation-based Simulation

In this technique, performance information is annotated into the application code and executed at the native environment. During the native execution, the previous annotated information is used to calculate the application performance. One example of annotation-based performance estimation is [9]. The C source code of the application is first lowered to an executable intermediate representation (IR). A set of machine independent optimizations, such as constant propagation, constant folding and dead code elimination are performed to remove redundant operations. The optimized IR is then analyzed to estimate the operation cost of each basic block. These costs are then annotated back to the IR, which in turn are natively compiled and executed to estimate the performance of the application. The performance estimation from a cycle accurate virtual prototype is exported to a concurrent functional simulator in [11]. Target binaries are simulated on cycle-accurate simulators to obtain timing information. This timing information is annotated back to the original C code at source line level. Finally, the SystemC simulation is performed on annotated code for fast performance estimation.

2.4 Hybrid Simulation

This technique combines the advantages of ISS, such as applicability to arbitrary programs and accurate estimation, with native execution of selected parts of the application. Native execution refers to the execution of a program directly on a simulation host and is typically much faster than ISS.

The pioneer work in this category is [13]. It proposes a hybrid performance estimation technique for single processors. Some parts of an application are executed

on the native host machine, whereas the rest runs on an ISS. Natively executed parts are the most frequently executed portion of the code. Since native execution is much faster than ISS, significant simulation speed is achieved.

The HySim framework [8] combines native execution with ISS similar [13]. However, it generates the C code containing performance information from the original C source code, similar to the annotation-based techniques [11]. It analyzes the source code of the application, and annotates operation cost and memory accesses. These annotations are evaluated at runtime to generate performance information in terms of processor cycles and memory latencies.

2.5 Comments on Existing Simulation Techniques

Due to system complexity, the ISS acceleration techniques such as dynamically compiled simulation, statically compiled simulation or JIT compiled simulation were not enough to achieve the desired simulation speed. Consequently, the partial simulation techniques, such as sampling and tracing, were proposed. The major drawback of sampling-based technique is that a large amount of pre-processing is needed for discovering the phases of the target application. The proposed performance estimation methodology in this article does not require to identify the regions of a program that are selectively simulated in detail while fast-forwarding over other portions. Therefore, no pre-processing is required.

The problem with trace-driven simulation is that the generated traces might become excessively large. Another issue is that trace-driven simulation relies on post-processing and cannot provide performance information at runtime. Annotation-based simulation techniques [9–11] provide simulation speedup as compared to pure ISS but still suffer with some restrictions. For example, the approach in [9] is applicable for RISC like processors and does not support super-scalar or VLIW architectures. The technique in [11] does not fully parse the C code. Similarly, developing a binary-to-C translator requires considerable efforts in [10]. The proposed approach in this article does not estimate the performance of the entire application. Instead, we use the simulation to estimate the performance of function blocks before the design space exploration loop.

Although simulation speedup is achieved in hybrid simulation techniques [8, 13], the major concern is the selection of application functions for native execution. For example, the limitation of [13] is that a training phase is required to build a procedure level performance model for the natively executed code. Similarly in [8], functions for native execution must contain no target dependent optimization. Our technique does not impose any restriction on the application code as the complete application is executed on the ISS of the target architecture.

3 Proposed Design Space Exploration

Section 2 provided state-of-the-art in reducing the simulation speed for fast DSE. We highlighted some major limitations in the existing approaches. The subsequent parts of this article will present a cycle-accurate performance estimation technique in a DSE framework. First, this sections presents the proposed DSE framework. Then, Sect. 4 will describe the performance estimation technique.

The proposed framework is shown in Fig. 2. It contains five stages: application specification, cycle-accurate performance estimation, computation architecture selection, code partitioning and communication architecture selection.

Application Transformation:

The application description is given in the form of reference C code. The proposed DSE framework starts by transforming a reference sequential code into composition of functional blocks. An application transformation methodology is presented in [16]. The application behavior may consist of hardware blocks written in RTL specifications (hardware IPs) or software blocks written as C functions. Hardware IPs are provided with their performance values. However, the performance of a software IP can not be determined a priory. We estimate the performance of software function blocks by performing simulations at cycle-accurate level in the second step of the proposed DSE framework which is the main concern of this article.

Performance Estimation:

The proposed DSE framework consists of two inner design loops: computation architecture selection loop and communication architecture selection loop. Before entering into computation architecture selection loop, it is necessary to estimate the

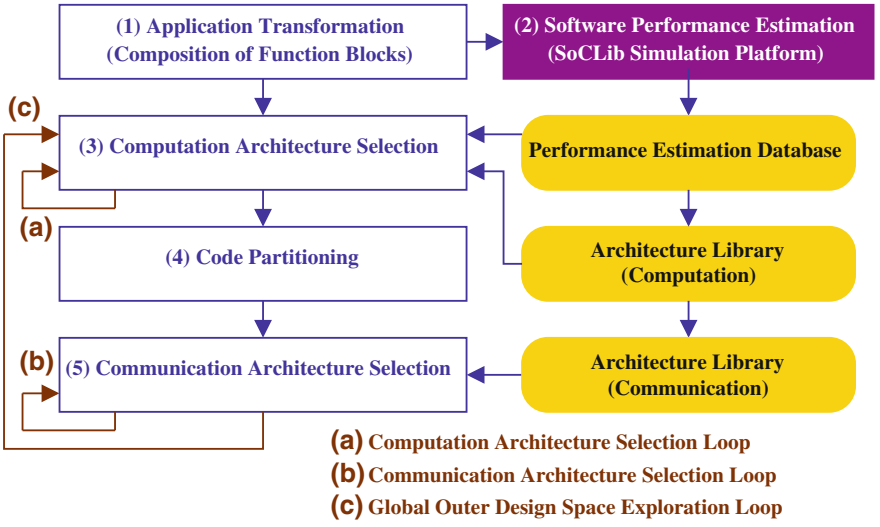


Fig. 2 Proposed design peace exploration framework

performance of software function blocks. A performance estimation technique will be described in Sect. 4 to perform simulations at cycle-accurate level.

Cycle-accurate simulation results of individual functional blocks are stored in a performance estimation database. The information in this database is used by the computation architecture selection loop in two different ways. First, it is used for component selection and mapping decision for each function block. Second, the performance of the entire application is estimated as a linear combination of block performances on the mapped components. In case the performance information of a function block is already recorded in the database, there is a trade-off whether we estimate the block performance again for a new application or not.

Computation Architecture Selection:

The third step performs computation architectural selection design loop. The inputs to this step are: application specification, performance estimation database contents and an initial architecture. It performs: (1) appropriate processing element selection, (2) mapping function blocks to the selected processing element and (3) evaluation of the estimated performance to check whether the given time constraints are met. In this step, a very abstract notion of communication overhead is used and is computed as the product of fixed cost and the number of data samples. This is because the communication architecture has not been determined yet.

Code Partitioning:

Once the mapping decision is made, the code for each processing element is synthesized in the fourth step. HW/SW co-simulation is performed to obtain memory traces from all processing components. The memory traces include both local and shared memory accesses from all processing elements. Memory traces are classified into three categories: code memory, data memory and shared memory. Code and data memories are associated with local memory accesses while shared memories are associated with inter-component communication.

Communication Architecture Selection:

The fifth step performs second inner design loop of the DSE framework. Based on the memory trace information generated by processing elements, the communication architectural selection loop selects the optimized communication architecture.

Outer Design Space Exploration Loop:

A global DSE loop updates the communication cost after the communication architecture is determined from the communication architectural selection loop. The key benefits of separating the computation from the communication are lower time complexity and extensibility.

4 Performance Estimation Technique

Section 3 presented the DSE framework with five stages. This section will present the second step of the proposed DSE framework by presenting a performance estimation methodology at functional level of the embedded software.

Basic Principle:

Application behavior specified in function blocks is instrumented and cross-compiled on ISS of the target processor. We assume that all architectural features of the processor are accurately modeled in the simulator. The compiled code is simulated on the simulation platform at cycle-accurate level to obtain the run time profile of each function block. It includes number of execution cycles and memory access counts. From the run time profile, we determine the representative performance values and store it in a performance estimation database. Once we have the performance estimation values for individual blocks, the performance of the entire application is computed as a linear combination of function block performance values.

Cyclic Dependency Problem:

The software performance estimation depends upon two things: compiler options and architecture features. Depending upon the compilation options, the performance variation can be as large as 100 %. Even though, function block performances were already recorded in the performance estimation database, we have to examine which compiler options were used before using performance values. The most important architecture feature is the memory system. If a cache system is used, cache miss rate and miss penalty, both affect the software performance.

As a result, there is a cyclic dependency between the performance estimation and the DSE. The system architecture is determined after the DSE but the performance estimation is required before the DSE. However, the accurate performance estimation is only possible after system architecture is determined. For example, memory access time is dependent on the communication architecture and memory system. This cyclic dependency is shown in Fig. 3.

Solution to Cyclic Dependency Problem:

This problem is solved by specifying the performance value of a software block on a processor with not a number but a pair: (CPU time, memory access counts). The CPU time is obtained from simulation assuming that the memory access cycle is 1 (perfect memory hypothesis). We record the memory access counts separately as the second element of the performance pair. Then, the block performance on a specific architecture will be the sum of the CPU time and the memory access counts multiplied by the memory access cycles. Memory access latency is defined from the

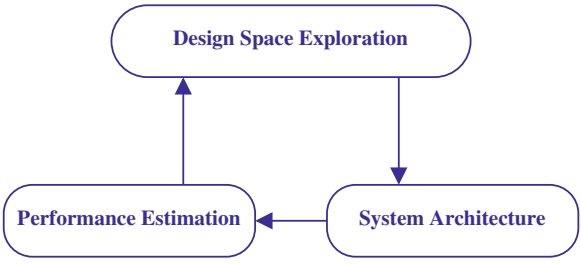


Fig. 3 Cyclic dependency problem

architecture and its value can be updated after performing the fifth step, which is the communication architecture selection loop, in the proposed DSE framework shown in Fig. 2. Such separation of memory access counts breaks the cycle dependency between the performance estimation and the design space exploration.

Data-Dependant Performance Estimation:

In addition to compiler options and architecture features, performance value of a software function block depends on input data. It is observed that the worst case takes much longer than the average case behavior. Therefore, for each function block, simulation is performed more than once with different input data. As a result, WCET and ACET are computed for each function block.

Definition 1 *Worst Case Execution Time (WCET)*—the maximum execution time of a function during multiple simulations with different input data. It should not be confused with the conventional meanings of the WCET used in static analysis. There is no guarantee of the worst case performance because we use the inputs that are not exhaustive in any sense. It just says that the performance is no worse than this value with high probability.

Definition 2 *Actual Case Execution Time (ACET)*—the average execution time of a function block during multiple simulations with different input data. Performance results with the WCET can be too pessimistic. ACET may reveal more realistic performance results for average case optimization.

However, it might be a problem to use the ACET as the performance measure for real time applications due to non-uniform execution time.

Definition 3 *Non-uniform Execution Time*—the phenomenon of executing different frames in a video sequence with different execution time. The in-depth discussion of non-uniform execution time will be presented in Sect. 6.4.

Performance Estimation with Real Test Data:

For estimating the performance of each block, a common method is to build a test bench program where a test vector generator provides input argument values to the function. This method has two serious drawbacks. First, it is very laborious to build a separate test bench program and analysis environment for each function block. Second, good test vectors are not easy to define.

The proposed approach overcomes these drawbacks by running the entire application. Since the entire application is already given at the specification stage, no additional effort of building a separate simulation environment is needed. And test vectors to the function blocks are all real, better than other synthetic test vectors. Since we are using the real test vectors for a function block, the average case performance value is meaningful when computing the average performance of the entire application by summing up the performance values of function blocks.

In order to measure the performance estimation, we obtain the number of processor clock cycles for a particular task execution. For more precise performance estimation, we can also measure the information related to the caches associated to a processor (number of cache Miss and Hit).

To summarize, the main features are:

- The performance value of a software block on an architecture component is specified as a pair: (CPU time, memory access counts).
- The proposed technique simulates the function blocks with different data set to obtain WCET and ACET for each function block.

4.1 Performance Estimation Database

The estimated performance information is recorded in a performance estimation database. There are multiple entries for each (functional block, processing component) pair depending upon the compiler options. Even with a given compiler option, a function block may have a different performance value at each execution because its performance is data dependent. For data dependent performance variations, we compute the WCET (Definition 1) and the ACET (Definition 2). The WCET and ACET results are stored in a performance estimation database.

Also, we need to distinguish the functional block performances by block parameters. For example, the execution time of an FIR filter is proportional to the number of filter taps. Different sets of filter coefficients are defined as block parameters while the same block definition is used. A block parameter that has an effect on the block performance is called a *factor* of the block. In short , the following tuple is updated to the performance estimation database.

(function name, processing element, compiler options, memory reads, memory writes, WCET, ACET)

The performance results in the performance estimation database is used by the computation architecture selection loop in two different ways. First, it is used for component selection and mapping decision for individual function blocks in the application. Second, the performance of the entire application is estimated as a linear combination of block performances on the mapped components.

4.2 Application Performance Estimation

This section explains performance estimation during computation architecture selection loop of the design space exploration framework in Fig. 2. The entire application performance is estimated as a linear combination of block performances for each candidate system architecture and mapping decision. Let:

- *"Fk"* be the name of the function such that for function *"F1"*, the value of *"k"* is equal to 1, for function *"F2"* the value of *"k"* is equal to 2 and so on.
- *"Pi"* be the name of the mapped component such that for component *"P1"*, *"P2"*, *"P3"*,. . . *"Pn"* the value of *"i"* is equal to 1, 2, 3 . . .,n.

- *"T(k, i)"* be the CPU time taken by function *"Fk"* mapped on component *"Pi"*.
- *"M(k)"* and *"I(k)"* be the number of memory accesses and the number of invocations for function *"Fk"* respectively.
- *"C(k, l)"* be the communication requirement of function *"Fk"* to the next function block *"Fl"*.
- *"N(m)"* be the memory access overhead of the selected candidate architecture.
- *"N(c)"* be the channel communication overhead of the selected architecture.

Then, the estimated performance of the entire application becomes,

$$\sum I(k) * \{T(k,i) + M(k) * N(m) + C(k,l) * N(c)\} \quad (1)$$

The accuracy of the estimated performance, provided in Eq. 1, depends on the accuracy of each term. For example, the value of "N(m)" and "N(c)" can be updated after performing communication architecture selection loop. Accuracy is also dependent on modeling of the candidate processing element.

Another cause of inaccuracy may come from the cache behavior. When the initial state of cache is different, the simulated cache behavior is also different, to make the performance estimation inaccurate. It also affects the number of memory access counts.

The third term in Eq. 1 may be included in the second term if the communication is performed through memory and asynchronous protocol is used. Otherwise, we need to pay extra overhead of synchronization and/or communication activities, indicated by the last term of Eq. 1.

For example, if an application has four functions F1, F2, F3 and F4 such that: (a) the functions F1 and F2 are mapped to processing element (PE) P1 and (b) functions F3 and F4 are mapped to processing element P2 as shown in Table 2.

By putting the values in Eq. 1, we obtain that the total execution cycles for function F1, F2, F3 and F4 are 325, 100, 300 and 320 respectively. A linear combination of these values will give the total number of execution cycles of the complete application.

This section described the basic principles of proposed performance estimation technique. Cyclic dependency between performance estimation and design space exploration for architecture selection was solved by specifying the performance value of a software function on an architecture component with a pair: (CPU time, memory access counts). Moreover, the proposed technique satisfied the three requirements for software performance estimation. The results of individual function blocks were stored in a performance estimation data base (Sect. 4.1). Performance of the entire application was computed by Eq. 1 in Sect. 4.2.

Table 2 Example of performance estimation with Eq. 1

Functions	PEs	T(k, i)	I(k)	M(k)	N(m)	C(k, L)	N(c)
F1	P1	30	5	5	5	5	2
F2	P1	35	2	3	5	0	2
F3	P2	25	5	5	4	5	3
F4	P2	40	4	10	4	0	3

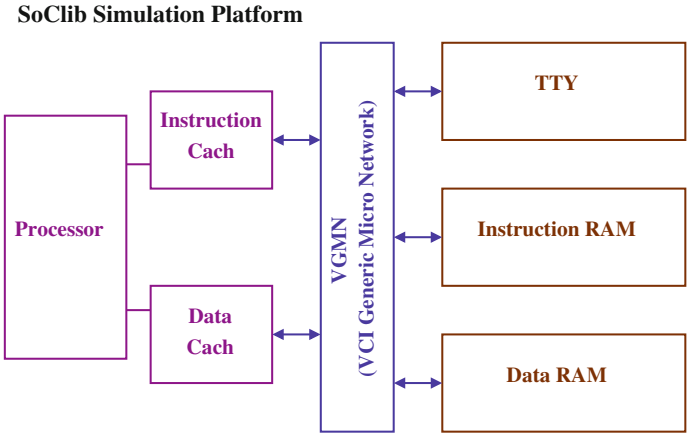


Fig. 4 SoCLib simulation platform

5 SoCLib Simulation Platform

Section 4 presented a performance estimation technique at cycle-accurate level. In order to implement the proposed technique, a simulation platform is required. We have chosen SoCLib [21] simulation platform for our experiments with H.264 application in Sect. 6. SoCLib is a library of open-source SystemC simulation modules. Example of simulation models available in SoCLib are processor models like *"PowerPC"*, *"ARM"*, *"MIPS"*, standard on-chip memories and several kinds of networks-on-chip. The *"VCI"* communication protocol is used to interface between IPs. In order to realize a simulation platform, components are chosen from a database of SystemC modules.

5.1 Realization of SoCLib Simulation Platform

A simulation platform is obtained by direct instantiation of hardware modules as shown in Fig. 4. A processor is used with its associated data and instruction cache. Standard memories such as instruction RAM and data RAM are used for storing the program and data respectively. A VCI Generic Micro Network (VGMN) is used to communicate between different components of the simulation platform. A dedicated component is used for displaying output (referred as TTY).

In order to realize a simulation platform, we write the top module *"top.cpp"* file which contains all required SoCLib component definitions (e.g. Processor, RAM, TTY etc). We define a mapping table to simplify the memory map definitions and the hardware component configurations before instantiating any hardware components. Mapping table itself is NOT a hardware component and it is used by the platform

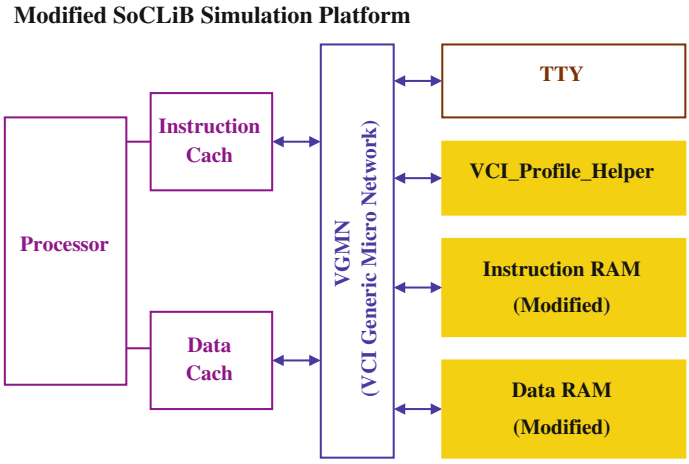


Fig. 5 Modified SoCLib simulation platform

designer to describe the memory mapping and address decoding scheme of any shared memory architecture build with the SoCLib hardware components. All the required SoCLib components are added to mapping table. We then create all the required components and associated signals and connect components and signals. Finally we define *"sc_main"* in the top module, and run simulation.

5.2 Modified Simulation Platform: An Extension of SoCLib

The information required for the proposed performance estimation framework can not be extracted through the already available tools in the SoCLib library. Therefore, modifications in various modules of the SoCLib platform or creation of new modules is required. In order to implement the proposed performance estimation methodology, we create a new module *"VCI_Profile_Helper"* and modify the existing module for instruction and data RAM. It allows to extract the required performance information which is not possible through already available tools in SoCLib [21]. In an open source framework like SoCLib, it is easy to make these changes. However, it may require some development time to modify the existing module or creating new ones.

The purpose of the new module *"VCI_Profile_Helper"* is to count number of execution cycles for a function block on a processing element during simulation. The purpose of modification in instruction and data RAMs is to generate a memory access profile for extraction of the information such as: time at which the request is made, the address of the transaction and the type of transfer etc. This information is stored in a simple text file during simulation. We modify the simulation platform such that the new simulation platform instantiate the component *"VCI_Profile_Helper"* and modified RAMs as shown in Fig. 5.

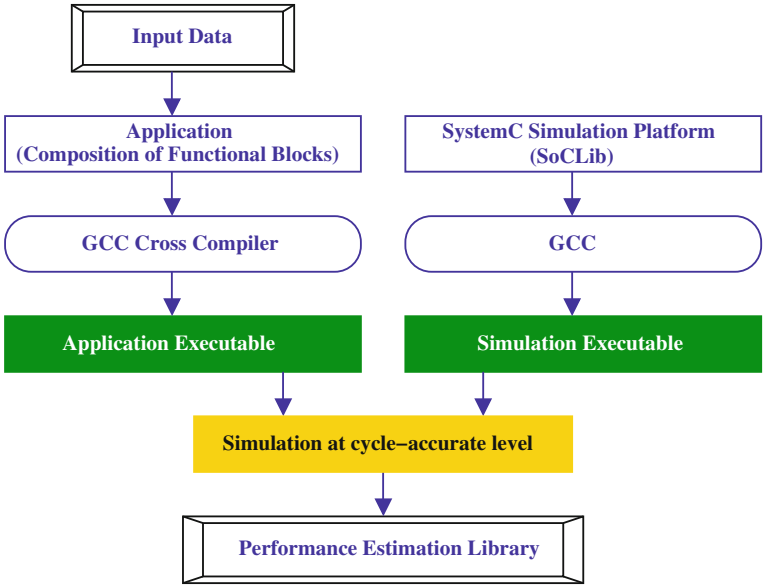


Fig. 6 Simulation flow using SoCLib simulation platform

5.3 Simulation Flow Using SoCLib Simulation Platform

Figure 6 represents the simulation flow that will be used in our experiments (Sect. 6). Application is compiled with the GNU GCC tool suite. SoCLib simulation platform models all architecture features. It is compiled with the GCC and yields a binary executable named *"simulation.x"*. The results of the simulation at cycle-accurate level for each function with different input data are stored in performance estimation database. Since the platform simulator models all architecture features and the binary executable is obtained after compilation. In addition to this, simulation is performed with different types of data set. Therefore, the simulation flow incorporates the three basic requirements of performance estimation.

This section described a simulation platform to implement the proposed performance estimation technique. The next section will use this simulation platform to perform experiments with H.264 video encoding application.

6 Experimental Results

This section presents experimental results for the performance estimation technique (Sect. 4) by using the SocLib simulation platform (Sect. 5) with X264 application which is an open source implementation of H.264.

6.1 Experimental Setup

The target simulation platform and simulation flow are shown in Figs. 5 and 6 respectively. In the simulation platform of Fig. 5, we have used cycle-accurate simulation models of different processors from SoCLib library. It includes ARM, PowerPC and MIPS with associated instruction and data cache. We encode 745 frames of QCIF format moving picture. The frame sequence consists of one I-type frame and the subsequent 734 P-type frames. We estimate the performance of each function block on PowerPC405, ARM7TDMI and ARM966 processors.

6.2 Performance Estimation on PowerPC405 Processor

The performance estimation results of X264 video encoder for PowerPC405 with 32 KB of instruction and data cache are summarized in Table 3. The first column of Table 3 lists all function blocks in the application. For each function block, simulations are performed with different data sets to obtain WCET (Definition 1) and ACET (Definition 2) listed in the second and the third columns respectively.

The diversity *"D"* is shown in the fifth column and is obtained by dividing the WCET with ACET. Total execution time (TET) for each function block is shown in the fourth column and is obtained by multiplying execution time (WCET or ACET depending upon the value of *"D"* for the function block) with total number of calls. We observe that the value of *"D"* is comparatively large for the *mc_chroma* block, *get_ref* block and the *IDCT* block. Therefore, using the WCET for these function blocks is not adequate measure of estimated performance for cost sensitive designs.

If the cache miss penalty is zero, which implies perfect cache hypothesis or no external memory access, the processor times become the performance of function

Table 3 Execution cycles of X264 video encoder

Name	WCET	AET	TET	D	Reads	Writes
Functions for SATD	8,993	8,150	1.05×10^{11}	1.10	528	802
Functions for SAD	3,596	3,129	1.6×10^{10}	1.15	928	1,578
get_ref	28,355	16,423	1.1×10^{11}	1.72	1,125	510
mc_chroma	29,259	17,517	7.9×10^{10}	1.67	1,844	729
Intra prediction	2,041	1,777	5.3×10^{9}	1.15	376	543
Functions for DCT	2,541	2,330	2.5×10^{9}	1.1	1,953	967
Functions for IDCT	2,253	750	9×10^{8}	3.0	837	458
Functions for Q	1,165	997	2.1×10^{9}	1.17	775	513
Functions for IQ	830	755	9.8×10^{8}	1.1	540	423
Entropy encoding	2,179	1,866	4.1×10^{10}	1.16	568	784
Miscellaneous			8.6×10^{9}			
Total			3.7×10^{11}			

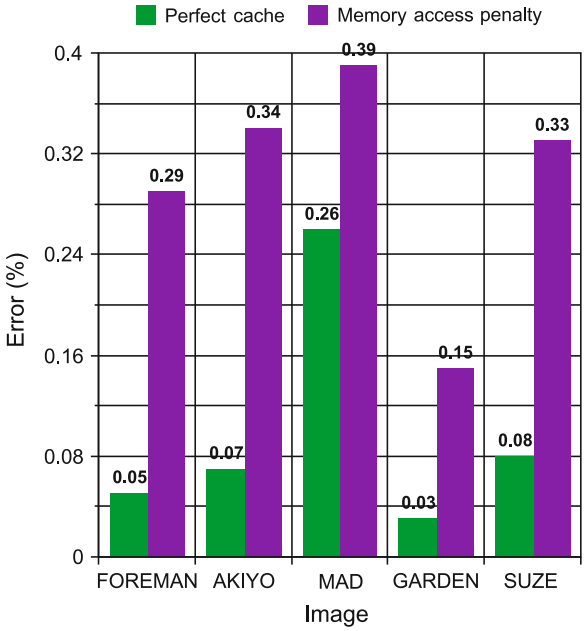


Fig. 7 Error of the estimated performance

blocks. Therefore, the linear combination of block performances results in a value of 378, 316, 262, 400 cycles and shown as 3.7×10^{11} in the last row of Table 3. The total execution time of the entire application without code augmentation results in a value of 377, 726, 530, 900 cycles. Note that the error between two results is just 0.155 %. It proves the accuracy of the proposed technique.

Now if we consider the cache miss penalty, the accuracy of the proposed technique is slightly degraded. The last two columns of Table 3 shows the total number of memory reads and memory writes for each function block that corresponds to cache misses. Figure 7 illustrates the error of the estimated performance obtained from Eq. 1 compared with the simulation results considering the cache miss penalty.

It shows that the error of the estimated performance is still under 0.5 % assuming that cache miss penalty is 5 bus cycles. It is important to note that frame sizes play an important role in the cache miss penalty. We can explain the effect of frame size on the cache miss penalty by the following example.

Suppose the memory system of the architecture is assumed to have two-level cache management. Usually, the size of the first level (L1) cache is small such that only a few data of reference frame can be cached, but the second-level (L2) cache is larger and it might be able to store the whole reference frame. Consider the scenario that the frame size is small such that the reference frame can be stored in the L2 cache. Under such a scenario, the cache miss penalty is equal to the access time of the L2 cache. In contrast, the cache miss penalty of the L1 cache is equal to that of

Table 4 Total execution time on different processors

Processor name	Total execution time
PowerPC405 (1)	3.7×10^{11}
PowerPC405 (2)	2.9×10^{11}
ARM7TDMI (1)	6.5×10^{11}
ARM7TDMI (2)	4.6×10^{11}
ARM966 (1)	5.2×10^{11}
ARM966 (2)	3.6×10^{11}

the L2 cache, if frame size is larger such that the required data cannot be obtained from the L2 cache. Therefore, the cache miss penalty of the L1 cache highly depends on the hit rate of L2 cache and the hit rate of the L2 cache is related to frame size.

Since all function blocks are executed in a single processor, there is no communication overhead included in this experiment. Figure 7 also shows the experimental results with other image samples. Here, we estimate the block performance separately for each image sample since the performance values are quite different depending on the scene characteristics.

6.3 Performance Estimation on Different Processors

Section 6.2 presented the experimental results on PowerPC405. This section shows the performance estimation of H.264 video encoder on different processors as shown in Table 4. As candidate processing elements, we have used PowerPC405, ARM7TDMI and ARM966 with an L1 cache only. The total number of execution cycles for one PowerPC405 processor, two PowerPC405 processors, one ARM7TDMI processor, two ARM7TDMI processors, one ARM966 processor and two ARM966 processors are 3.7×10^{11}, 2.9×10^{11}, 6.5×10^{11}, 4.6×10^{11}, 5.2×10^{11} and 3.6×10^{11} respectively.

6.4 Non-uniform Execution Time of Input Video Sequence

Performance estimation in Sect. 6.3 is made by taking the ACET or WCET depending upon the diversity between them. However, It might be a problem to use the ACET as the performance measure for real time applications. Figure 8 shows the variations of H.264 video encoder execution time. ACET is represented as "Ave" and the WCET is represented as "Max" in Fig. 8.

Depending upon the mode of operation, the execution time varies. This variation of execution time is due to the multiple ways of macroblock analysis in different video frames. If such variations in execution time is ignored in the computation of average execution time and the implementation barely accommodates the average performance, all P frames will result in deadline miss.

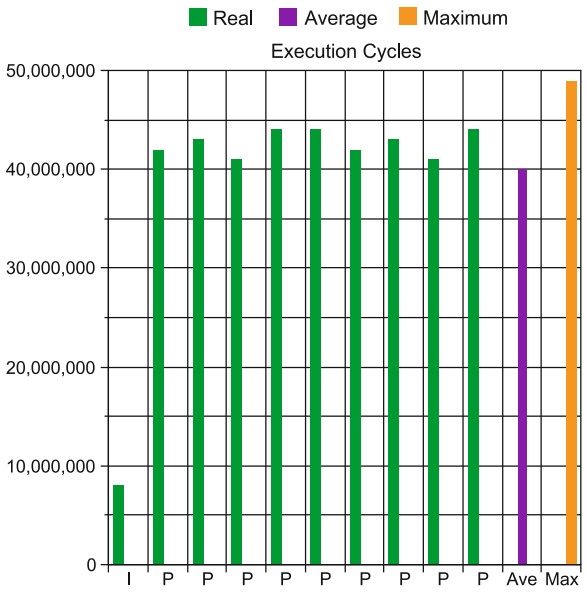


Fig. 8 The consecutive frames, average and worst case execution cycles

One solution to this problem is to use different performance measure instead of the ACET. As shown in Fig. 8, using the WCET is a costly solution. The more optimized solution is to record the execution time of each block in case of the worst case behavior of the whole application. Then, we compared the recorded execution time with the average execution time of each block and choose the maximum. In this way, no deadline miss occurs and we save 14 % of the estimated execution time compared with the worst case estimation.

6.5 Decrease in Simulation Time

The simulation time of the entire X264 video encoding application with PowerPC405 processor is 16 h and 30 min. This simulation time is recorded for a QCIF video of 745 frames with one I-type frame and the subsequent 734 P-type frames. Simulation is performed for all function blocks in the application. In order to reduce the simulation time, performance values of function blocks in performance estimation database are used as shown in Table 5.

Decrease in simulation time during each iteration is shown in the third column of Table 5. 25 % decrease is obtained by using "SATD" performance values from performance estimation database. Similarly, 30 % decrease is obtained by using "get_ref" values from performance estimation database and so on.

Table 5 Decrease in simulation time

Type of simulation	Simulation time	Decrease in simulation time (%)
Complete application	16 h and 30 min	–
SATD values from database	12 h and 20 min	25
get_ref values from database	11 h and 35 min	30
SATD and get_ref values	7 h and 25 min	55

Whether we estimate the performance of function block again for a new application or not is a trade-off. The performance of a function block may depend on what application it is used in and what are the input value ranges. Estimating the function block performance again with a new application gives more accurate information for the next design space exploration step. However, it costs time overhead of candidate processor. If the number of candidate processors are large, this overhead may be too huge to be tolerated within the tight budget of design time.

This section has presented the experimental results of the proposed performance estimation technique with X264 application. The PowerPC405, ARM7TDMI and ARM966 from SoCLib library were used to perform simulations. Experimental results included the cache miss penalty as well as the non-uniform execution time. Finally, the decrease in simulation time was illustrated in Table 5.

7 Conclusions

This article presented a DSE framework consisting of five stages, with the emphasis on software performance estimation at cycle-accurate level. The proposed performance estimation methodology stored performance estimation results of each function block on a simulation platform in a performance estimation database. The database values were used for architecture components selection.

After component selection and mapping decision was made, the performance of the entire application was computed as a linear combination of individual function blocks performance values. The proposed technique considered the effects of architecture features, compiler optimizations and data dependent behavior of the application. We have extended the SoCLib library to build a simulation platform for experiments.

Experimentation with H.264 encoder has proved that the proposed performance estimation technique satisfy the requirements of accuracy and adaptability at the same time. A simple linear combination of performance numbers has given an accurate (within 1 %) performance estimate of the entire application.

References

1. S. Abdi, Y. Hwang, L. Yu, G. Schirner, D.D. Gajski, Automatic TLM generation for early validation of multicore systems. IEEE Des. Test Comput. **28**(3), 10–19 (2011)
2. J. Aycock, A brief history of just-in-time. ACM Comput. Surv. **35**(2), 97–113 (June 2003)
3. I. Boandhm, B. Franke, N. Topham, Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator, in International Conference on Embedded Computer Systems (SAMOS), (2010), pp. 1–10
4. G. Braun, A. Hoffmann, A. Nohl, H. Meyr, *Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description, in Proceedings of the 14th International Symposium on System Synthesis (ISSS'01)* (Montreal, Quebec, Canada, October, 2001), pp. 57–62
5. D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0. Technical report CS-TR-1997-1342 (1997)
6. L. Eeckhout, K. De Bosschere, H. Neefs, *Performance analysis through synthetic trace generation, in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)* (Austin, Texas, USA, April, 2000), pp. 1–6
7. Evan Data Corporations, in Software Development Platforms-2011 Rankings (2011).
8. L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, H. Meyr, Multiprocessor performance estimation using hybrid simulation, in Proceedings of the 45nd Design Automation Conference (DAC'08). Anaheim, CA, USA **8–13**, 325–330 (June 2008)
9. K. Karuri, M.A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid, H. Meyr, *Fine-grained application source code profiling for ASIP design, in Proceedings of the 42nd Design Automation Conference (DAC'05)* (Anaheim, California, USA, June, 2005), pp. 329–334
10. M.T. Lazarescu, J.R. Bammi, E. Harcourt, L. Lavagno, M. Lajolo, *Compilation-based software performance estimation for system level design, in Proceedings of the IEEE International High-Level Validation and Test Workshop* (Washington, DC, USA, November, 2000), pp. 167–172
11. T. Meyerowitz, A. SangiovanniVincentelli, M. Sauermann, D. Langen, *SourceLevel timing annotation and simulation for a heterogeneous multiprocessor, in Proceedings of the Design, Automation and Test in Europe Conference (DATE'08)* (Munich, Germany, March, 2008), pp. 276–279
12. M.H. Rashid, in System Level Design: A Holistic Approach. (Lap Lambert Academic Publishing, 2011).
13. A. Muttreja, A. Raghunathan, S. Ravi, N.K. Jha, Hybrid simulation for energy estimation of embedded software. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(10), 1843–1854 (2007)
14. J. Park, S. Ha, Performance analysis of parallel execution of H.264 encoder on the cell processor, in Proceedings of 5th IEEE/ACM Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'07), Salzburg, Austria, October 2007, pp. 27–32
15. W. Qin, J. D'Errico, X. Zhu, A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation. 4th International Conference on Hardware/software Co-design and System Synthesis (CODES+ISSS'06), Seoul, Korea, October 2006, pp. 193–198.
16. M. Rashid, F. Urban, B. Pottier, *A transformation methodology for capturing data flow specification, in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'09)* (Innsbruck, Austria, February, 2009), pp. 220–225
17. M. Reshadi, N. Dutt, P. Mishra, A retargetable framework for instruction-set architecture simulation. ACM Trans. Embed. Comput. Syst. **5**(2), 431–452 (2006)
18. M. Rosenblum, E. Bugnion, S. Devine, S. Herrod, Using the SimOS machine simulator to study complex computer systems. ACM Trans. Model. Comput. Simul. **7**(1), 78–103 (1997)
19. M.S. Suma, K.S. Gurumurthy, Fault simulation of digital circuits at register transfer level. Int. J. Comput. Appl. **29**(7), 1–5 (2011)
20. T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003)

21. Soclib, Simulation Platform, www.soclib.fr
22. R.E. Wunderlich, T.F. Wenisch, B. Falsafi, J.C. Hoe, *SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, in Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)* (San Diego, USA, June, 2003), pp. 84–95

Multicast Algorithm for 2D de Bruijn NoCs

Reza Sabbaghi-Nadooshan, Abolfazl Malekmohammadi and Mohammad Ayoub Khan

Abstract The performance of the network is measured in terms of throughput. The throughput and efficiency of interconnect depends on network parameters of the topology. Therefore, topology of any communication networks has an important role to play for efficient design of network. The De Bruijn topology has the potential to be an interesting option for future generations of System-on-Chip (SoC). Two-dimensional (2-D) de Bruijn is proposed for Networks-on-Chips (NoCs) applications. We can improve performance in the two dimensional Bruijn NoCs by improvement of routing algorithm. In this chapter, we have proposed a multicast routing algorithm for 2-D de Bruijn NoCs. The proposed routing algorithm is compared with unicast routing using Xmulator under various traffics conditions. Based on comparison results, the proposed routing has significantly improved the performance and power consumption of the NoC in comparison with unicast routing under light and moderate traffic loads in hot spot and uniform traffics with various message lengths.

1 Introduction

With recent advances in VLSI technologies, modern chips can embed large number of processing cores as a multi-core chip. Such multicore chips require efficient communication architecture to provide a high performance connection between the cores. Network-on-Chip (NoC) has been recently proposed as a scalable communication

R. Sabbaghi-Nadooshan (✉) · A. Malekmohammadi
Electrical Engineering Department, Islamic Azad University Central Tehran Branch, Tehran, Iran
e-mail: R_sabbaghi@iauctb.ac.ir

A. Malekmohammadi
e-mail: Malekmohammadi.abolfazl@gmail.com

M. A. Khan
Department of Computer Science and Engineering, Sharda University, Greater, Noida, India
e-mail: ayoub@ieee.org

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_9, © Springer-Verlag Berlin Heidelberg 2014

architecture for multicore chips [1]. In NoC paradigm, every core communicates with other cores using on-chip channels and an on-chip router. On-chip channels construct a predefined structure called topology. The NoC is a communication centric interconnection approach which provides a scalable infrastructure to interconnect different IPs and sub-systems in a SoC [2]. The NoC can make SoC more structured, reusable and can also improve their performance. Since the communication between the various processing cores will be deciding factor for the performance of such systems, therefore we need to focus on making this communication faster as well as more reliable.

Also, the network topology has direct impact on important NoC parameters e.g., network diameter, bisection width, and the routing algorithm [3]. The topology has a great impact on the system performance and reliability. It generally influences network diameter (the length of the maximum shortest path between any two nodes), layout and wiring [4]. These characteristics mainly determine the power consumption and average packet latency [5]. Before we delve deeper into the widely used topologies, the main characteristics of network topology which are described in Table 1 should be understood first.

Authors have proposed several topologies in the literature such as mesh topology [6], hypercube topology [7], tree topology [8], and de Bruijn topology. Each of these topologies has its pros and cons; for example, mesh topology is used in the fabrication of several NoCs because of its simple VLSI implementation; however, other topologies are also favored by NoC designers due to their exclusive features. The de Bruijn topology is one of those topologies which provide a very low diameter in comparison with the mesh topology, however imposes a cost equal to a linear

Table 1 Characteristics of network topology

Characteristics	Description
Bisection of network	A bisection of a network is a cut that partitions the entire network nearly in half
Throughput	The throughput of a network is the data rate in bits per second that the network accepts per input port. The ideal throughput is defined as the throughput assuming a perfect routing and flow control i.e. Load is balanced over alternate paths and no idle cycles on bottleneck channels
Latency	The latency of the network is the time required for a message to traverse a network, i.e. a time taken for a packet, flit or message to reach from source to destination. It also includes the time taken for computing arbitration logic as well as routing computation and other delays
Diameter	The diameter (D) of a network is the maximum internodes distance. The smaller the diameter of a network, the less time it takes to send a message from one node to the farthest node
Node degree	The node degree is defined as the number of physical channels emanating from a node. This attribute shows the node's I/O complexity

array topology. The De Bruijn topology is a well-known structure which is initially proposed [9] for parallel processing networks. Several researchers have studied topological properties [10], routing algorithms [11, 12], VLSI layout efficiency [10] and other aspects of the de Bruijn networks [13]. NoC designers also favor to the de Bruijn topology, since it provides logarithmic diameter and cost equal to a linear array topology [13].

Considering the reputation of the mesh topology and the low network diameter of de Bruijn topology, de Bruijn can be inspired mesh-based topology for NoCs. In our previous work [14] we have suggested two dimensional Bruijn for NoCs, and the proposed topology has better performance relation to mesh. However, we can improve performance in the two dimensional Bruijn NoCs. We have used three-dimension layout or torus as we used in [15, 16]. Furthermore, we can improve routing algorithm. In this chapter, we use multicast routing for the improvement of performance.

2 Multicast Routing Algorithm

2.1 The de Bruijn Topology

Since the proposed network topology is based on de Bruijn, this section briefly introduces the de Bruijn topology [17, 18]. An n-dimensional de Bruijn topology is a directed graph including k^n nodes. In this topology, node $u = (u_n, \ldots, u_1)$ is connected to the node $v = (v_n, \ldots, v_1)$ if and only if $u_i = v_{i+1} \quad 1 \leq i \leq n-1$. In other words, node v has a directed link to node u if and only if

$$u = v \times k + r (\mathrm{mod}\, k^n), \quad 0 \leq r \leq k-1 \tag{1}$$

According to definition of de Bruijn topology, in-degree and out-degree of all nodes is equal to k. Therefore, the degree of each node is equal to $2k$. In addition, the diameter of de Bruijn topology is equal to n which is optimal. Owing to the fact that these connections are unidirectional, the degree of the network is the same as a one-dimensional mesh network (or linear array network). The diameter of a de Bruijn network with size N, that is, the distance between nodes 0 and $N-1$, is equal to $\log(N)$.

In a de Bruijn network the two operations namely shuffle operation and shuffle-exchange operation are defined as follows to ease the routing algorithm in this network. In the shuffle operation, address of the current node i.e., $v = (v_{n-1}, \ldots, v_0)$ is logically rotated by one bit in the left direction. In the shuffle-exchange operation, address of the current node i.e., $v = (v_{n-1}, \ldots, v_0)$ is logically rotated by one in the left direction and then the least significant bit is complemented. Consider a $k = 2$ de Bruijn network as shown in Fig. 1. Using the shuffle operation, node 1 goes to node 2, and using the shuffle-exchange operation, node 1 goes to node 3. Using these

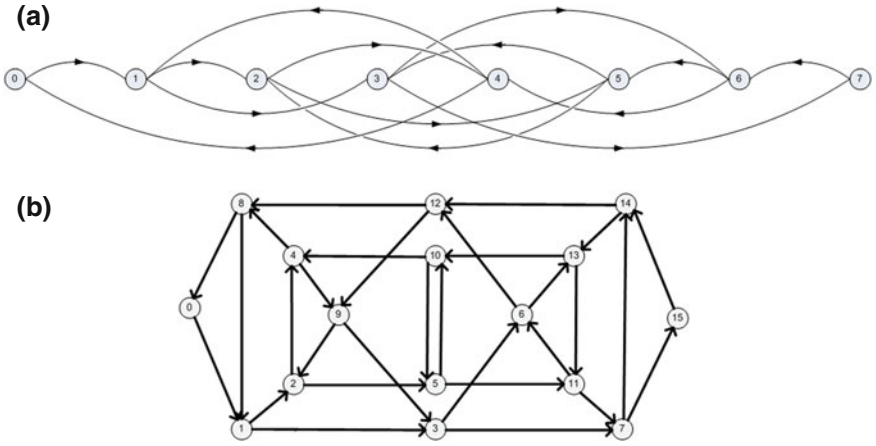


Fig. 1 The de Bruijn network with (**a**) 8 nodes and (**b**) 16 nodes

two operations, the following routing algorithm can deliver any packet in a de Bruijn network.

For routing algorithm, Ganesan [11] splits the de Bruijn networks into two trees i.e., T1 and T2, (see Fig. 2) to perform the routing with at most four steps. At first, the message is routed between T1 to T2 if it is necessary, and then in T2, and then the message is routed between T2 to T1 and finally in T1. A two-dimensional de Bruijn topology is a two-dimensional mesh topology in which nodes of each dimension form a de Bruijn network. An 8×8 two-dimensional de Bruijn is shown in Fig. 3.

The proposed routing algorithm for two-dimensional de Bruijn exploits two trees T1 and T2 in each dimension of the network. Like XY routing in mesh networks, the deterministic routing first applies the routing mechanism in rows to deliver the packet to the column at which the destination is located. Afterwards, the message is routed to the destination by applying the same routing algorithm in the columns.

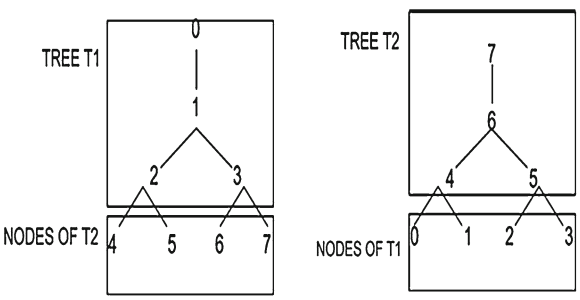


Fig. 2 Trees T1 and T2 for N = 8

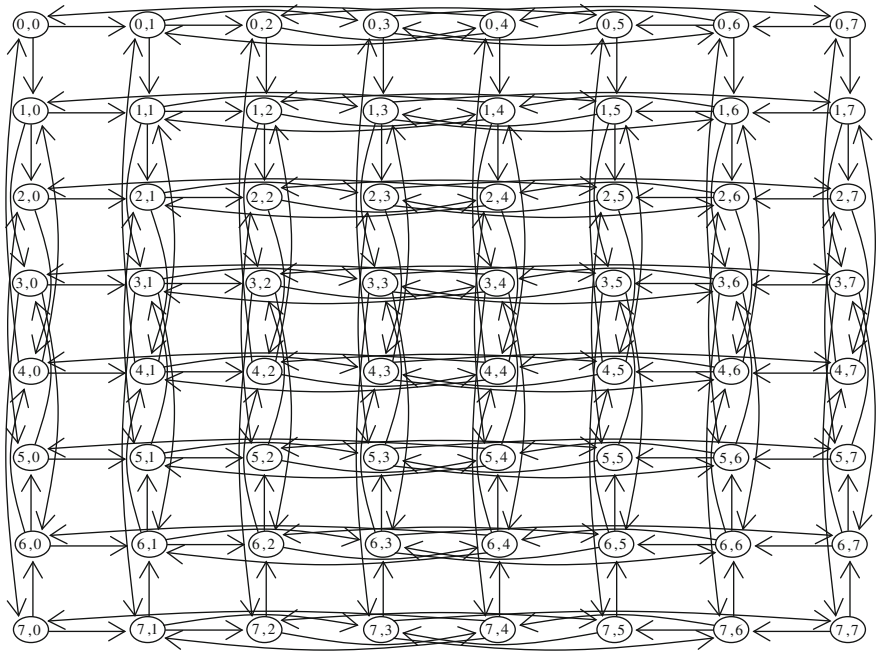


Fig. 3 A two-dimensional de Bruijn with 64 nodes composed from eight 8-node de Bruijn networks (as shown in Fig. 1a) along each dimension

2.2 The Proposed Multicast Algorithm

Deterministic routing is based on minimum hop and distance between source and destination nodes, unlike partially adaptive and fully adaptive is fixed and is shown with d_{S-D}. In deterministic routing, if the specific node k is a part of the route, number of hops between nodes of source S and destination D equal to the distance between source node and specific node k plus the distance between node k and destination node and vice versa. Above condition will be

$$d_{S-D} = d_{S-k} + d_{k-D} \tag{2}$$

If a specific node k is next bode (N), then the above condition will be as follows:
Condition 0:

$$d_{S-D} = d_{S-N} + d_{N-D} \tag{3}$$

where d is number of hops between two nodes S is source node D is destination node N is next node.

In this chapter, all nodes that are necessary for routing source S to destination D, are shown with $P(S, D)$. Also main route is the route that source node moves to marked destination node.

In proposed multicast routing (that is based on minimum distance between nodes of source and destination); a destination is selected randomly (D_0) and marked. Message routes as unicast to deliver it to the marked destination (D_0). At each hop, N (that is next node in current message) and D_P (that is one of destination nodes except marked destination node in current message) are placed in condition (0). If condition 0 for specific destination (D_P) is true, the next node in main route belongs to $P(S, D_P)$. Therefore, message is not duplicated and routing is continued with a message. Otherwise, next node in main route with next node in $P(S, D_P)$ is different and for routing D_P, message should be duplicated. Therefore, a necessary condition to duplicate the message is below condition.

Condition 1:

$$d_{S-D_p} \neq d_{S-N} + d_{N-D_P} \tag{4}$$

where d is number of hops between two nodes S is source node D_P is one of destination nodes (except the marked destination node in current message) N is next node in current message.

Condition (1) is not sufficient to duplicate the message because when it is true for a specific destination D_P, for next steps, will remain true and message will duplicate at each hop to routing D_P frequently but for routing each destination node, only one message is needed. Therefore, condition (2) (that prevents to copy the repeated message) is necessary.

Condition 2:

$$d_{S-D_P} = d_{S-C} + d_{C-D_P} \tag{5}$$

where d is number of hops between two nodes S is source node D_P is one of destination nodes (except the marked destination node in current message) C is current node in current message.

Conditions (1) and (2) check whose next node and current node in main route belongs to $P(S, D_P)$ respectively. Actually, in two conditions, last common node between main route and $P(S, D_P)$ determine to duplicate the message.

If condition (1) and condition (2) for D_P are true simultaneously, the message should duplicate to routing D_P (and D_P is marked). In other words, the message is duplicated if and only if, current node between two routes is the common and next node is different in same two routes. Above steps for all messages into a network perform until the number of messages into network equal to number of destinations.

2.3 Giving an Example

As an example, we suppose that node (3, 0) has message for nodes (5, 4), (7, 5) and (1, 1) as Fig. 4.

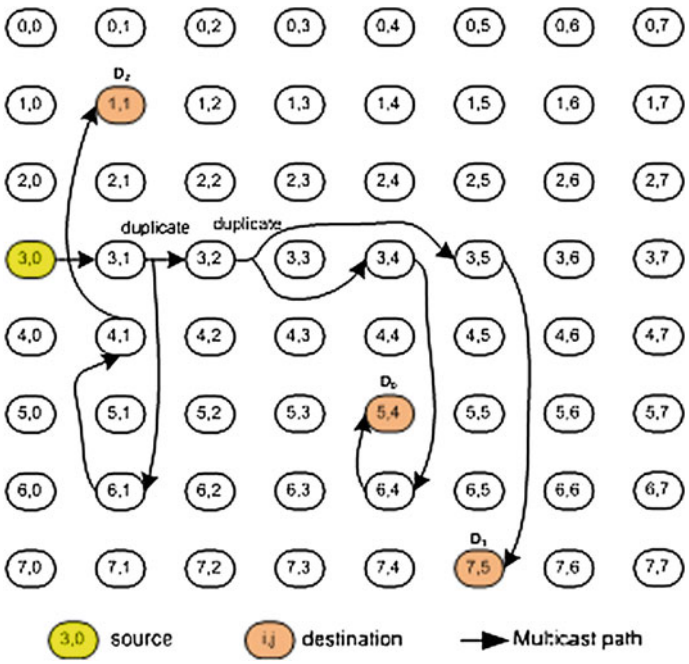


Fig. 4 Tree-based multicast algorithm based on unicast XY routing

For unicast routing we have:

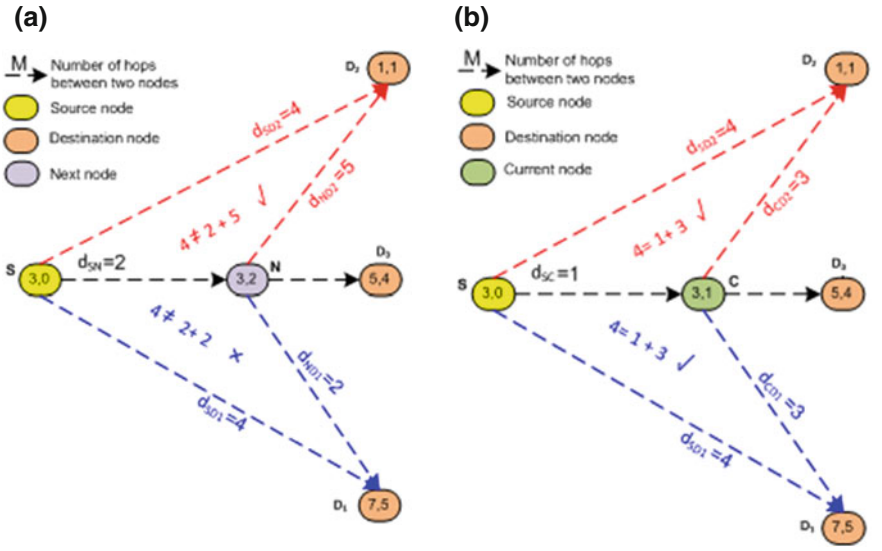
$$(3,0) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (3,4) \rightarrow (6,4) \rightarrow (5,4)$$
$$(3,0) \rightarrow (3,1) \rightarrow (3,2) \rightarrow (3,5) \rightarrow (7,5)$$
$$(3,0) \rightarrow (3,1) \rightarrow (6,1) \rightarrow (4,1) \rightarrow (1,1)$$

For multicasting, one destination is selected randomly (d0 : (5,4)) and message is routed as unicast $R((3,0),(5,4))$. At each hop, two above conditions (1), (2) for the remaining destinations ((7, 5), (1, 1)) are checked and if both of conditions are true, message will be duplicated.

In first step (Fig. 5), condition (1) is false for both of destinations (7, 5), (1, 1) but second condition is true for them. Therefore, the next node (3, 1) in first path is shared with two other paths. Now, there is only one message into the network.

$$S = (3,0), \mathrm{C} = (3,0), N = (3,1), D = \{(7,5),(1,1)\}$$

In second step as Fig. 6, condition (2), (1) are true for destination (1, 1). Thus, message will be duplicated to route (1, 1) and in this new message destination node (1, 1) is marked. However, for destination (7, 5), according to condition (1) that is false, message will not duplicate. Now, there are two messages into the network.



(c)

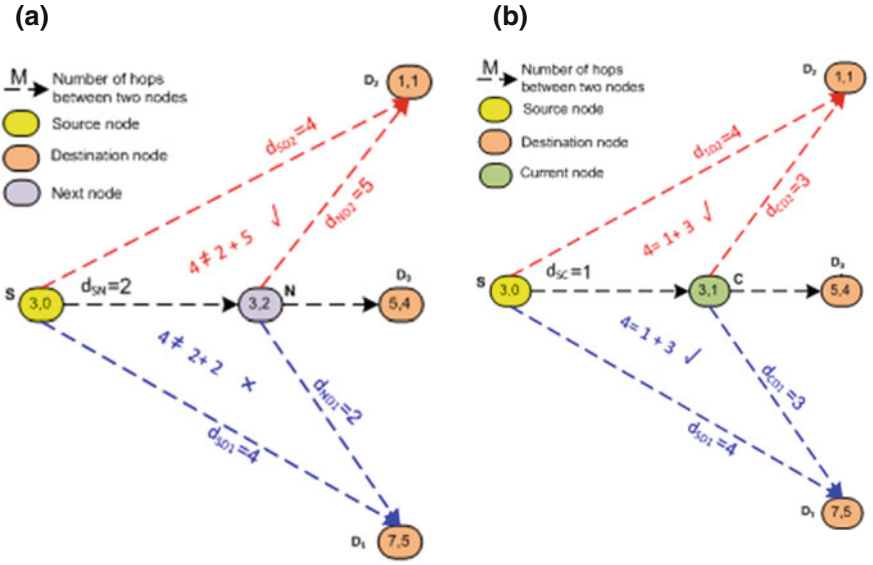
	d1	d2
Condition1	false	false
Condition2	True	True
Copy msg	false	false

Fig. 5 First hop checking. **a** the $d_{SDi} \neq d_{SN}+d_{NDi}$ (condition 1), **b** $d_{SDi} = d_{SC}+d_{CDi}$ (condition 2), **c** Table

$$S = (3, 0), C = (3, 1), N = (3, 2), D = \{(7, 5), (1, 1)\}$$

In third step (Fig. 7), condition (1), (2) are true for destination (7, 5) and message will be duplicated to route it and (7, 5) is marked in this new message. Even so condition (2) for destination (1, 1) is false. Therefore, message will not copy. Now, there are three messages into the networks.

After this step (when the number of destinations equal to number of messages), at all next steps, condition (2) for all destinations will be false and no message will be duplicated. Furthermore, routing will be as unicast. Pseudo code of proposed algorithm is shown in Fig. 8.

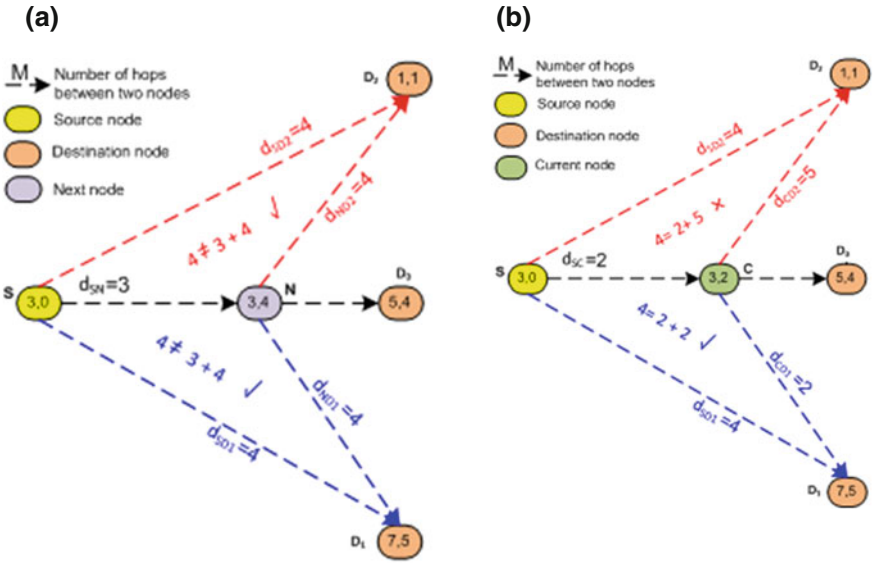


	d1	d2
Condition1	false	True
Condition2	True	True
Copy msg	false	True

Fig. 6 Second hop checking. **a** the $d_{SDi} \neq d_{SN} + d_{NDi}$ (condition 1), **b** $d_{SDi} = d_{SC} + d_{CDi}$ (condition 2), **c** Table

3 Simulation Results

To evaluate the performance of suggested routing, we develop a discrete event simulator operating at the flit level using xmulator [19]. We set the networks link width to 128 bits. Each link has the same bandwidth and one flit transmission is allowed on a link. The power is calculated based on a NoC with 65 nm technology whose routers operate at 2.5 GHz. We set the width of the IP cores to 1 mm, and the length of each wire is set based on the number of cores it passes. The number of virtual channels is two and maximum of simulation events are 15,000,000. The simulation results are obtained for 8 × 8 de Bruijn NoCs with XY routing algorithm, using the routing algorithms described in the previous section. The message length is assumed to be 32 and 64 flits and messages are generated according to a Poisson distribution with rate λ. The traffic pattern can be *Uniform* and *Hotspot* [20].



(c)

	d1	d2
Condition1	True	True
Condition2	True	false
Copy msg	True	false

Fig. 7 Third hop checking. **a** the $d_{SDi} \neq d_{SN} + d_{NDi}$ (condition 1), **b** $d_{SDi} = d_{SC} + d_{CDi}$ (condition 2), **c** Table

In the following figures the average message latency and power consumption are shown. The *x*-axis of these figures indicates the generation rate and *y*-axis indicates power and delay in our simulations. Figure 9 compares the average message latency for different traffic patterns with different message lengths of 32 and 64 flits. As can be seen, the multicast routing has smaller average message latency with respect to the unicast routing algorithm for the full range of network load under various traffic patterns (especially in uniform traffic). For hotspot traffic load a hotspot rate of 16 % is assumed (i.e. each node sends 16 % of messages to the hotspot node (node (7, 7)) and the rest of messages to other nodes uniformly). As can be seen in the figures, the multicast routing can better cope with non-uniformity of the network traffic and its performance improvement over unicast under hotspot traffic pattern.

Figure 10 demonstrates power consumption of the multicast routing and unicast routing with various traffic patterns. Simulation results indicate that the power

```
Distance (source, current node, next node, destination):
Calculates the number of hops between nodes of source-
destination (d_SD), current -destination (d_CD), source-
current (d_SC) and next node-destination(d_ND).
Next node: it is after current node
des: set of destinations
Input:  multicast set : <source,{D_0,D_1,D_2,…..,D_n}>
1: Created messages=0
2: While (des ≠ ∅)
3: select a destination from des randomly and marked it.
4:   Created messages++
5:       While (created messages > 0)
6:             If (current node == any one des)
7:        des=des-{marked destination in current message}
8:               Created messages - -
9:                 End if
10:                If (created messages==0)
11:  select a destination from des randomly and marked it
12:                       Created messages++
13:                End If
14:      For all created messages
15:               For all destinations (D_i) ∈  des-{marked
destination in current message}
16: //according to current and destination nodes in mes-
sage, next node is available.
17:        distance (source, current node, next node, D_i )
18:             If ((d_SD i <> d_SN + d_ND i) && (d_SD i =  d_SC + d_CD i))
//d_SN=d_SC+1
19:         copy a message for routing D_i
20:           Marked D_i
21:              Created messages++;
22:           Route as unicast from current node to D_i
(marked destination)
23:              End If
24:          End For
25:     End For
26:      End While
27: End While
```

Fig. 8 Multicast pseudo code

consumption of multicast routing is less than the power dissipated of unicast routing for light to medium traffic loads. However, it begins to behave differently near heavy traffic regions where the unicast routing saturates and cannot handle more traffic.

Obviously, handling more traffic load (after the point that the unicast is saturated) requires more power for multicast routing. Note that when the unicast routing

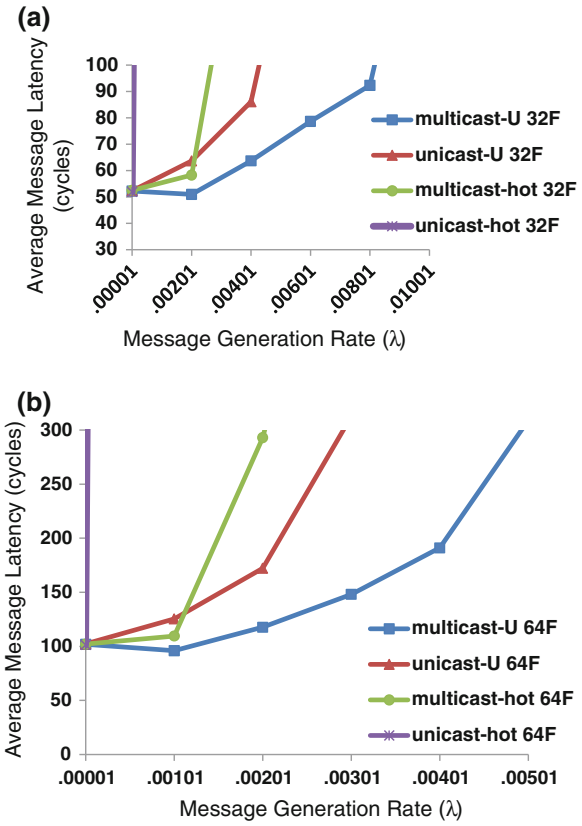


Fig. 9 Compares the latency for different traffic patterns with different message lengths of 32 and 64

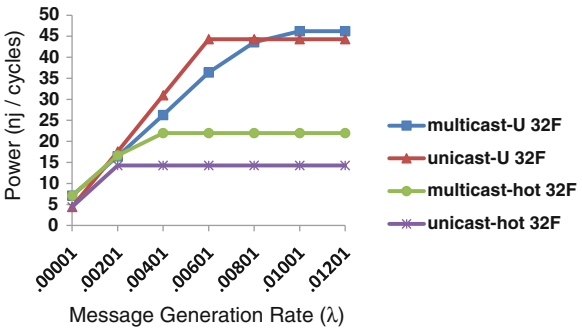


Fig. 10 Demonstrates power consumption for various traffic patterns and message lengths of 32 and 64

approaches its saturation region, the multicast routing can still handle the traffic effectively and the saturation point for them is higher than that unicast routing.

In Fig. 11a, the average message latency is plotted as a function of message generation rate at each node for the multicast routing and unicast routing for different message lengths of 32 and 64 flits. As can be seen in the Fig. 11a, the multicast routing has smaller average message latency with relation to the unicast routing. Figure 11b compares the total network power in different message lengths of 32 and 64. The obtained result of xmulator indicates the multicast routing goes to the saturation later and can send more packages; therefore has more power.

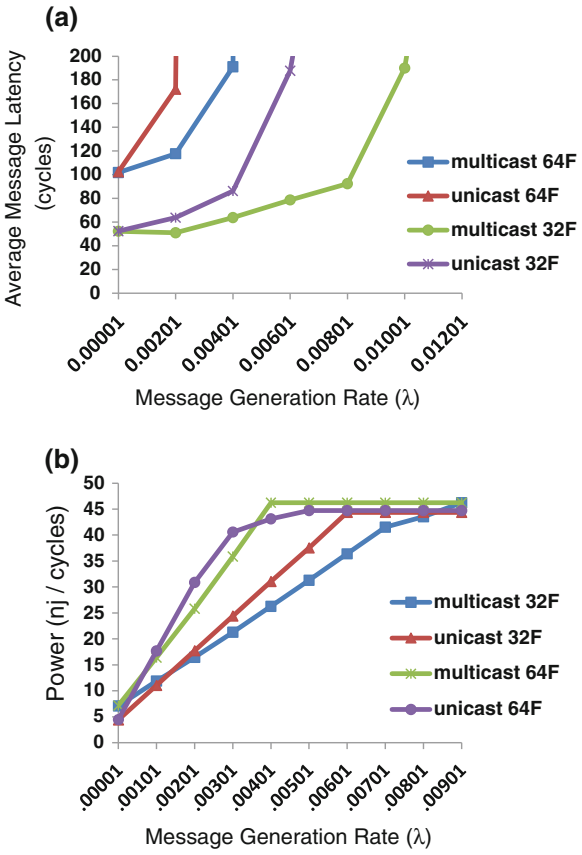


Fig. 11 Compares the performance and total network power in different message lengths of 32 and 64

4 Conclusion

Two-dimensional de Bruijn topology and routing algorithm is proposed for NoC. There are many methods to improve the performance of de Bruijn topology. One of them is improvement of routing algorithm. This chapter has proposed multicast routing algorithm for two-dimensional de Bruijn NoCs. The algorithms are compared using Xmulator simulator. Simulation experiments were conducted to assess the network latency and power consumption of the proposed routing. Results obtained shows the proposed routing has improved in terms of performance and power consumption of the NoC in comparison with unicast routing under light and moderate traffic loads in hot spot and uniform traffics with various message lengths.

For the future work, the fault tolerant can be used in multicast routing algorithm. Furthermore, this multicast method can be used for other digraph networks such as two-dimensional shuffle-exchange networks [21].

References

1. A. Jantsch, H. Tenhunen, *Network on Chip* (Kluwer Academic Publishers, Dordrecht, 2003)
2. W.J. Dally, H. Aoki, Deadlock-free adaptive routing in multicomputer networks using virtual channels. IEEE Trans. Parallel Distrib. Syst. **4**, 466–475 (1993)
3. P.P. Pande, C. Grecu, M. Jones, A. Ivanov, R. Saleh, Performance evaluation and design trade-offs for network-on-chip interconnect architectures. IEEE Trans. Comput. **54**, 1025–1040 (2005)
4. E. Salminen, A. Kulmala, T.D. Hamalainen, Survey of network-on-chip proposals. OCP-IP white paper (2008)
5. A. Flores, J.L. Aragon, M.E. Acacio, An energy consumption characterization of on-chip interconnection networks for tiled CMP architectures. J. Supercomputing **45**, 341–364 (2008)
6. U.Y. Ogras, R. Marculescu, Application-specific network-on-chip architecture customization via long-range link insertion. IEEE/ACM International Conference on Computer Aided Design (2005)
7. M. Chatti, S. Yehia, C. Timsit, S. Zertal, A hypercube-based NoC routing algorithm for efficient all-to-all communications in embedded image and signal processing applications. HPCS 623–630 (2010)
8. H. Matsutani, M. Koibuchi, Y. Yamada, H. Amano, Fat H-tree: a cost-effective tree-based on-chip-networks. IEEE Trans. Parallel Distrib. Syst. **20**, 1126–1141 (2009)
9. N.G. de Bruijn, A combinatorial Problem. Koninklijke Nederlands Akad. van Wetenschappen Proc. **49**, 758–764 (1946)
10. M.R. Samanathan, D.K. Pradhan, The de Bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI. IEEE Trans. Comp. **38**, 567–581 (1989)
11. E. Ganesan, D.K. Pradhan, Wormhole routing in de Bruijn networks and Hyper-de Bruijn Networks, in IEEE International Symposium on Circuits and Systems (ISCAS) (2003), pp. 870–873
12. H. Park, D.P. Agrawal, A novel deadlock-free routing technique for a class of de Bruijn based networks, in 7th IEEE Symposium on Parallel and Distributed Processing (1995), pp. 92–97
13. M. Hosseinabady, J. Mathew, D.K. Pradhan, Application of de Bruijn graphs to NoC. DATE (2007), pp. 111–116
14. R. Sabbaghi-Nadooshan, M. Modarressi, H. Sarbazi-Azad, The 2d DBM: an attractive alternative to the mesh topology for network-on-chip, in IEEE International Conference on Computer Design (2008), pp. 486–490

15. R. Sabbaghi-Nadooshan, M. Modarressi, H. Sarbazi-Azad, The 2D digraph-based NoCs: attractive alternatives to the 2D mesh NoCs. J. Supercomputing **49**, 1–21 (2012)
16. R. Sabbaghi-Nadooshan, H. Sarbazi-Azad, The kautz mesh: a new topology for SoCs, in International Conference on SoC Design (2008), pp. 300–303
17. G.P. Liu, K.Y. Lee, Optimal routing algorithms for generalized de Bruijn digraph, inInternational Conference on Parallel Processing (1993), pp. 167–174
18. J. Mao, C. Yang, Shortest path routing and fault-tolerant routing on de Bruijn networks. Networks **35**, 207–215 (2000)
19. A. Nayebi, S. Meraji, A. Shamaei, H. Sarbazi-Azad, Xmulator: a listener-based integrated simulation platform for interconnection networks, in Proceedings of Asian International Conefference on Modelling and Simulation (2007), pp. 128–132
20. J. Duato, S. Yalamanchili, L.M. Ni, *Interconnection Networks*. Morgan Kaufman (2003)
21. R. Sabbaghi-Nadooshan, M. Modarressi, H. Sarbazi-Azad, 2D SEM: a novel high-performance and low-power mesh-bases topology for networks-on-chip. Int. J. Parallel Emergent Distrib. Syst. **25**, 331–344 (2010)

Functional and Operational Solutions for Safety Reconfigurable Embedded Control Systems

Atef Gharbi, Mohamed Khalgui and Mohammad Ayoub Khan

Abstract The chapter deals with run-time automatic reconfigurations of distributed embedded control systems following component-based approaches. We classify reconfiguration scenarios into four forms: (1) additions-removals of components, (2) modifications of their compositions, (3) modifications of implementations, and finally (4) simple modifications of data. We define a new multi-agent architecture for reconfigurable systems where a Reconfiguration Agent which is modelled by nested state machines is affected to each device of the execution environment to apply local reconfigurations, and a Coordination Agent is proposed for any coordination between devices in order to guarantee safe and coherent distributed reconfigurations. We propose technical solutions to implement the whole agent-based architecture, by defining UML meta-models for agents. In the execution scheme, a task is assumed to be a set of components having some properties independently from any real-time operating system. To guarantee safety reconfigurations of tasks at run-time, we define service and reconfiguration processes for tasks and use the semaphore concept to ensure safety mutual exclusions. We apply the priority ceiling protocol as a method to ensure the scheduling between periodic tasks with precedence and mutual exclusion constraints.

A. Gharbi (✉) · M. Khalgui
INSAT, Tunis, Tunisia
e-mail: atef.elgharbi@gmail.com

M. Khalgui
e-mail: khalgui.mohamed@gmail.com

M. A. Khan
Sharda University, Gr. Noida, India
e-mail: ayoub@ieee.org

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_10, © Springer-Verlag Berlin Heidelberg 2014

1 Introduction

Nowadays, the new generations of distributed embedded control systems are more and more sophisticated since they require new forms of properties such as reconfigurability, reusability, agility, adaptability and fault-tolerance. The first three properties are offered by new advanced component-based technologies, whereas the last two properties are ensured by new technical solutions such as multi-agent architectures.

New generations of component-based technologies have recently gained popularity in industrial software engineering since it is possible to reuse already developed and deployed software components from rich libraries. A Control Component is a software unit owning data of the functional scheme of the system. This advantage reduces the time to market and allows minimizations of the design complexity by supporting the system's software modularity. This chapter deals with run-time automatic reconfigurations of component-based applications by using multi-agent solutions. An agent is assumed to be a software unit allowing the control of the system as well as its environment before applying automatic reconfigurations. The reasons for which reconfigurations may be taken are classified into two categories [33]: (1) corrective reasons: if there is one component which is misbehaving, then it is automatically substituted by a new one which is assumed to run correctly. The new component is supposed to have the same functionalities as the old one. (2) Adaptive reasons: even the component-based application is running well, dynamic adaptations may be needed as a response to the new environment evolutions, in order to extend new functionalities or to improve some required functional properties. Dynamic reconfigurations can cover the following issues: (1) architecture level which means the set of components to be loaded in memory to constitute the implemented solution of the assumed system; (2) control level which means the compositions of components; (3) implementation level which means the behavior of components encoded by algorithms; and (4) data level which means the global values. We define a multi-agent architecture for reconfigurable embedded control systems where a Reconfiguration Agent is affected to each device of the execution environment to apply automatic reconfigurations of local components, and a Coordination Agent which is used for coordination between distributed Reconfiguration Agents in order to allow coherent distributed reconfigurations. The Coordination Agent is based on a coordination protocol using coordination matrices which define coherent simultaneous reconfigurations of distributed devices. We propose useful meta-models for Control Components and also for intelligent agents. These meta-models are used to implement adaptive embedded control systems. As we choose to apply dynamic scenarios, the system should run even during automatic reconfigurations, while preserving correct executions of functional tasks.

Given that Control Components are defined in general to run sequentially, this feature is inconvenient for real-time applications which typically handle several inputs and outputs in a too short time constraint. To meet performance and timing requirements, a real-time must be designed for concurrency. To do so, we define at the operational level some sequential program units called real-time tasks.

Thus, we define a real-time task as a set of Control Components having some real-time constraints. We characterize a task by a set of properties independently from any Real Time Operating System (RTOS). We define service processes as software processes for tasks to provide system's functionalities, and define reconfiguration processes as tasks to apply reconfiguration scenarios at run-time. In fact, service processes are functional tasks of components to be reconfigured by reconfiguration processes. To guarantee a correct and safety behavior of the system, we use semaphores to ensure the synchronization between processes. We apply the famous algorithm of synchronization between reader and writer processes such that executing a service is considered as a reader and reconfiguring a component is assumed to be a writer process. The proposed algorithm ensures that many service processes can be simultaneously executed, whereas reconfiguration processes must have exclusive access. We study in particular the scheduling of tasks through a Real Time Operating System. We apply the priority ceiling protocol proposed by Sha et al. [49] to avoid the problem of priority inversion as well as the deadlock between the different tasks. The priority ceiling protocol supposes that each semaphore is assigned a priority ceiling which is equal to the highest priority task using this semaphore. Any task is only allowed to enter its critical section if its assigned priority is higher than the priority ceilings of all semaphores currently locked by other tasks.

In this chapter, we continue our research by proposing an original implementation of this agent-based architecture. We assume that agent controls the plant to ensure the system running physically. The design and the implementation of such agent under Real-Time constraints are the scope of this study. The main contributions of this chapter are the following: (1) a complete study of Safety Reconfigurable Embedded Control Systems from the functional level (i.e. dynamic reconfiguration system with a multi-agent system) to the operational level (i.e. decomposition of the system into a set of tasks with time constraints); (2) a global definition of real-time task with its necessary parameters independently from any real-time operating system; (3) the scheduling of these real-time tasks considered as periodic tasks with precedence and mutual exclusion constraints. To our best of knowledge, there is no research works which deal with these different points together.

We present in Sect. 2 the state of art about dynamic reconfiguration. Section3 presents the benchmark production systems FESTO and EnAS that we follow as running examples in the chapter. We define in Sect. 4 a multi-agent architecture and the communication protocol to ensure safety in a distributed embedded control systems. Section 5 presents the real-time task model and studies the safety of its dynamic reconfiguration as well as the scheduling between the different tasks. We finally conclude the chapter in Sect. 6.

2 Dynamic Reconfiguration

The new generation of industrial control systems is addressing today new criteria as flexibility and agility [43, 48]. We distinguish two reconfiguration policies: *static* and *dynamic* policies such that static reconfigurations are applied off-line to

apply changes before any system cold start [3], whereas dynamic reconfigurations are dynamically applied at run-time. Two cases exist in the last policy: manual reconfigurations applied by users [47] and automatic reconfigurations applied by intelligent agents [2]. We are interested in automatic reconfigurations of an agent-based embedded control system when hardware or software faults occur at run-time. The system is implemented by different complex networks of Control Components. In literature, there are various studies about dynamic reconfigurations applied to component-based applications. Each study has its strength and its weakness. In the article [35], the authors propose to block all nodes involved in transactions (considered as sets of interactions between components) to realize dynamic reconfigurations. This study has influenced many research works later. Any reconfiguration should respect the consistency propriety which is defined as sets of logical constraints. A major disadvantage of this approach is the necessity to stop all components involved in a transaction. In the article [4], problem of dynamic reconfigurations in CORBA is treated. The authors consider that consistency is related to Remote Procedure Call Integrity. To ensure this property, they propose to block the incoming before the outgoing links. However, the connection between components must be acyclic in order to be able to block connections in the right order. A dynamic reconfiguration language based on features [41] is proposed. The authors use the control language MANIFOLD where processes are considered as black boxes having ports of communication. In this case, the communication is anonymous. The processes having access to shared data are connected in cyclic manners to wait tokens that visit each one at turn (as in token ring). Although the novelty of this solution, there is a loss of time especially at waiting until receiving the token to access to the shared data or also to reconfigure the system. Another study [46] is proposed to apply dynamic updates on graphical components (for example button, graphical interface, ...) in a .Net framework. To do so, the authors associate for each graphical component an appropriate running thread. The synchronization is ensured through the reader-writer-locks. The dynamic reconfiguration is based on blocking all involved connections. Due to rw-locks, this solution works only on local applications. In addition, they define [45] a new reconfiguration algorithm ReDAC (Reconfiguration of Distributed Application with Cyclic dependencies) ensuring dynamic reconfigurations in distributed systems to be based on running multi-threads. This algorithm is applied to capsules which are defined as groups of running components. As disadvantage, the proposed algorithm uses counter variables to count on-going method calls for threads which lead to consume further space memory and treatment time.

To our best of knowledge, there is no research works which treat the problem of dynamic software reconfigurations of component-based technology with semaphores. The novelty of this chapter is the study of dynamic reconfiguration with semaphore ensuring the following points: (1) blocking connections without blocking involved components; (2) safety and correctness of the proposed solution; (3) independence of any specific language; (4) verification of consistency (i.e. logical constraints) delegated to the software agent; (5) suitable for large-scale applications.

3 Benchmark Production Systems: FESTO and EnAS

We present two Benchmark Production Systems[1]: FESTO and EnAS available in the research laboratory at the Martin Luther University in Germany.

3.1 The FESTO System

The FESTO Benchmark Production System is a well-documented demonstrator used by many universities for research and education purposes, and it is used as a running example in the context of this chapter. FESTO is composed of three units: Distribution, Test and Processing units. The Distribution unit is composed of a pneumatic feeder and a converter to forward cylindrical work pieces from a stack to the testing unit which is composed of the detector, the tester and the elevator. This unit performs checks on work pieces for height, material type and color. Work pieces that successfully pass this check are forwarded to the rotating disk of the Processing unit, where the drilling of the work piece is performed. We assume in this research work two drilling machines *Drill_machine*1 and *Drill_machine*2 to drill pieces. The result of the drilling operation is next checked by the checking machine and the work piece is forwarded to another mechanical unit. In this research chapter, three production modes of FESTO are considered according to the rate of input pieces denoted by *number_pieces* into the system (i.e. ejected by the feeder).

- **Case 1**: **High production**. **If** $number\_pieces \geq Constant1$, **then** the two drilling machines are used at the same time in order to accelerate the production. In this case, the Distribution and the Testing units have to forward two successive pieces to the rotating disc before starting the drilling with *Drill_machine*1 **AND** *Drill_machine*2. For this production mode, the periodicity of input pieces is $p = 11$ s.
- **Case 2**: **Medium production**. **If** $Constant2 \leq number\_pieces < Constant1$, **then** we use *Drill_machine*1 **OR** *Drill_machine*2 to drill work pieces. For this production mode, the periodicity of input pieces is $p = 30$ s.
- **Case 3**: **Light production**. **If** $number\_pieces < Constant2$, **then** only the drilling machine *Drill_machine*1 is used. For this production mode, the periodicity of input pieces is $p = 50$ s.

On the other hand, if one of the drilling machines is broken at run-time, then we have to only use the other one. In this case, we reduce the periodicity of input pieces to $p = 40$ s. The system is completely stopped in the worst case if the two drilling machines are broken.

[1] Detailed descriptions are available in the website: http://aut.informatik.uni-halle.de.

3.2 The EnAS System

The Benchmark Production System EnAS was designed as a prototype to demonstrate energy-antarcic actuator/sensor systems. For the sale of this contribution, we assume that it has the following behavior: it transports pieces from the production system (i.e. FESTO system) into storing units. The pieces in EnAS shall be placed inside tins to close with caps afterwards. Two different production strategies can be applied: we place in each tin one or two pieces according to production rates of pieces, tins and caps. We denote respectively by nb_{pieces}, $nb_{tins+caps}$ the production number of pieces and tins (as well as caps) per hour and by *Threshold* a variable (defined in user requirements) to choose the adequate production strategy. The EnAS system is mainly composed of a belt, two Jack stations (J_1 and J_2) and two Gripper stations (G_1 and G_2). The Jack stations place new produced pieces and close tins with caps, whereas the Gripper stations remove charged tins from the belt into storing units. Initially, the belt moves a particular pallet containing a tin and a cap into the first Jack station J_1.

According to production parameters, we distinguish two cases,

- **First production policy**: **If** ($nb_{pieces}/nb_{tins+caps} \leq Threshold$), **then** the Jack station J_1 places from the production station a new piece and closes the tin with the cap. In this case, the Gripper station G_1 removes the tin from the belt into the storing station St_1.
- **Second production policy**: **If** ($nb_{pieces}/nb_{tins+caps} > Threshold$), **then** the Jack station J_1 places just a piece in the tin which is moved thereafter into the second Jack station to place a second new piece. Once J_2 closes the tin with a cap, the belt moves the pallet into the Gripper station G_2 to remove the tin (with two pieces) into the second storing station St_2.

4 Multi-agent System

We define a multi-agent architecture for distributed safety systems. Each reconfiguration agent is affected in this architecture to a device of the execution environment to ensure Functional Safety. Nevertheless, the coordination between agents in this distributed architecture is inevitable because any individual decision may affect the performance of the others. To guarantee safe distributed reconfigurations, we define the concept of *Coordination Matrix* that defines correct reconfiguration scenarios to be applied simultaneously in distributed devices and we define the concept of *Coordination Agent* that handles coordination matrices to coordinate between distributed agents. We propose a communication protocol between agents to manage concurrent distributed reconfiguration scenarios.

The communication protocol between agents respects the different following points: (1) The Reconfiguration agents control the plant constituted by several physical processes. (2) At the beginning, all the Reconfiguration agents are assigned

Table 1 The agent characteristics

Agent type	Percepts	Actions	Goals	Environment
Reconfiguration agent	Something needs an intervention	Reconfigure the plant	Safe state	Physical plant
Coordination agent	Reconfiguration request	Contact the other agents	Coordination between agents	The whole system

a specific reconfiguration. (3) The Reconfiguration agent controlling the system can not apply more than one reconfiguration at any time. (4) The Reconfiguration agent decides to apply a new reconfiguration if some conditions are verified. (5) The Reconfiguration may be applied in a local system (in this case, only the associated Reconfiguration agent is concerned) or in a distributed system (in this case, many Reconfiguration agents have to coordinate together to put the whole system in a safe state). (6) The Reconfiguration agent does not know if the other agents will cooperate to put the system into safe state. (7) At the reception of a reconfiguration request, the agent chooses one action from the available possibilities (accept or refuse). The Reconfiguration agent may refuse the request if it is not possible to apply this new reconfiguration. (8) An agent is called cooperative if it always accepts the reconfiguration request. An agent is called selfish if it always refuses the new reconfiguration.

Before introducing the communication protocol, we begin with presenting a Reconfiguration Agent as well as the coordination agent. To resume the characteristics of each one, the Table 1 presents the main information.

4.1 Software Architecture of Reconfiguration Agents

We propose an agent-based architecture to control embedded systems at run-time. The agent checks the environment's evolution and reacts when new events occur by adding, removing or updating Control Components of the system. To describe the dynamic behavior of an intelligent agent that dynamically controls the plant, we use nested state machines in which states correspond to finite state machines. A finite state machine can be defined as a state machine whose states, inputs and outputs are enumerated. The nested state machine is represented as the following:

$$\text{NSM} = (SM_1, SM_2, \ldots, SM_n)$$

Each state machine (SM_i) is a graph of states and transitions. A state machine treats the several events that may occur by detecting them and responding to each one appropriately. We define a state machine as the following:

$$SM_i = (S_i, S_{i0}, I_i, O_i, \textit{Pre-cond}_i, \textit{Post-cond}_i, t_i)$$

- $S_i = \{s_{i1}, \ldots, s_{ip}\}$: the states;
- S_{i0} the initial state;
- $I_i = \{I_{i1}, \ldots, I_{im}\}$: the input events;
- $O_i = \{O_{i1}, \ldots, O_{ik}\}$: the ouput events;
- $Pre\text{-}cond_i$: the set of conditions to be verified before the activation of a state;
- $Post\text{-}cond_i$: the set of conditions to be verified once a state is activated;
- $t_i : S_i \times I_i \rightarrow S_i$: the transition function.

We propose a conceptual model for a nested state machine in Fig. 1 where we define the classes *Nested State Machine*, *State machine*, *State*, *Transition*, *Event* and *Condition*. The *Nested State Machine* class contains a certain number of *State machine* classes. This relation is represented by a composition. The *Transition* class is double linked to the *State* class because a transition is considered as an association between two states. Each transition has an event that is considered as a trigger to fire it and a set of conditions to be verified. This association between the *Transition* class and *Event* and *Condition* classes exists and is modeled by the aggregation relation.

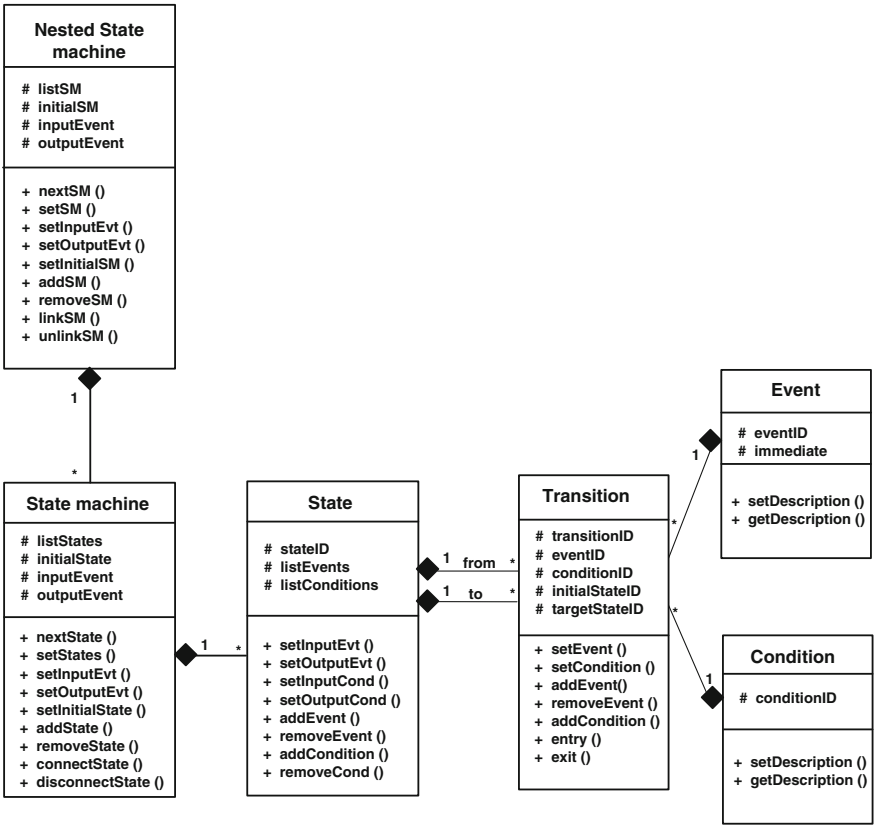


Fig. 1 The Meta-model nested state machine

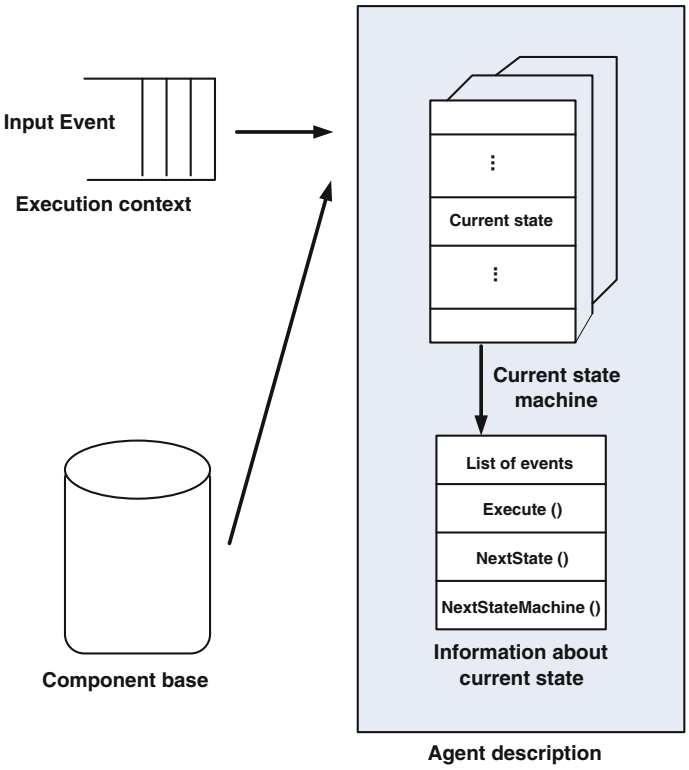


Fig. 2 The internal agent behavior

We propose a generic architecture for intelligent agents depicted in Fig. 2. This architecture consists of the following parts: (1) the Event Queue to save different input events that may take place in the system, (2) the intelligent software agent that reads an input event from the Event Queue and reacts as soon as possible, (3) the set of state machines such that each one is composed of a set of states, (4) each state represents a specific information about the system. The agent, based on nested state machines, determines the new system's state to execute according to event inputs and also conditions to be satisfied. This solution has the following characteristics: (1) The control agent design is general enough to cope with various kinds of embedded-software based-component application. Therefore, the agent is uncoupled from the application and from its Control Components. (2) The agent is independent of nested state machines: it permits to change the structure of nested state machines (add state machines, change connections, change input events, and so on) without having to change the implementation of the agent. This ensures that the agent continues to work correctly even in case of modification of state machines. (3) The agent is not supposed to know components that it has to add or remove in a reconfiguration case.

In the following algorithm, the symbol Q is an event queue which holds incoming event instances, *ev* refers to an event input, S_i represents a State Machine, and $s_{i,j}$ a state related to a State Machine S_i. The internal behavior of the agent is defined as follow:

1. the agent reads the first event *ev* from the queue Q;
2. searches from the top to the bottom in the different state machines;
3. within the state machine SM_i, the agent verifies if *ev* is considered as an event input to the current state $s_{i,j}$ (i.e. ev $\in$ I related to $s_{i,j}$). In this case, the agent searches the states considered as successor for the state $s_{i,j}$ (states in the same state machine SM_i or in another state machine SM_l);
4. the agent executes the operations related to the different states;
5. repeats the same steps (1–4) until no more event exists in the queue to be treated.

Algorithm 1: GenericBehavior

```
begin
while (Q.length() > 0) do
    ev ← Q.Head()
   For each state machine SM_i do
       s_{i,j} ← currentState_i
      If ev ∈ I(s_{i,j}) then
          For each state s_{i,k} ∈ next(s_{i,j})
           such that s_{i,k} related to S_i do
              If execute(s_{i,k}) then
                  currentState_i ← s_{i,k}
                  break
              end if
          end for
          For each state s_{l,k} ∈ next(s_{i,j})
           such that s_{l,k} related to S_l do
              If execute(s_{l,k}) then
                  currentState_l ← s_{l,k}
                  break
              end if
          end for
      end if
   end for
end while
end.
```

First of all, the agent evaluates the pre-condition of the state $s_{i,j}$. If it is false, then the agent exits, Else the agent determines the list of Control Components concerned by this reconfiguration, before applies the required reconfiguration for each one. Finally, it evaluates the post-condition of the state $s_{i,j}$ and generates errors whenever it is false.

```
    Function execute(s_{i,j}) : boolean
begin
    If ¬s_{i,j}.PreCondition then
        return false
    else
        listCC ← getInfo(s_{i,j}.info)
        For each CC ∈ listCC do
            CC.reconfigure()
        end for
        If ¬s_{i,j}.PostCondition then
            Generate error
        end if
        return true
    end if
end.
```

4.2 Communication Protocol

To guarantee safe distributed reconfigurations, we define the concept of *Coordination Matrix* that defines correct reconfiguration scenarios to be applied simultaneously in distributed devices and we define the concept of *Coordination Agent* that handles coordination matrices to coordinate between distributed agents.

Let *Sys* be a distributed safe system of n devices, and let $Ag_1, \ldots, Ag_n$ be n agents to handle automatic distributed reconfiguration scenarios of these devices. We denote in the following by $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ a reconfiguration scenario applied by Ag_a ($a \in [1, n]$) as follows: (1) the corresponding *ASM* state machine is in the state ASM_{i_a}. Let $cond^{a}_{i_a}$ be the set of conditions to reach this state, (2) the *CSM* state machine is in the state CSM_{i_a,j_a}. Let $cond^{a}_{j_a}$ be the set of conditions to reach this state, (3) the *DSM* state machine is in the state DSM_{k_a,h_a}. Let $cond^{a}_{k_a,h_a}$ be the set of conditions to reach this state. To handle coherent distributed reconfigurations that guarantee safe behaviors of the whole system *Sys*, we define the concept of *Coordination Matrix* of size (n, 4) that defines coherent scenarios to be simultaneously applied by different agents. Let *CM* be such a matrix that we characterize as follows: each line a ($a \in [1, n]$) corresponds to a reconfiguration scenario $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ to be applied by Ag_a as follows:

$$CM[a, 1] = i_a; \quad CM[a, 2] = j_a; \quad CM[a, 3] = k_a; \quad CM[a, 4] = h_a$$

According to this definition: **If** an agent Ag_a applies the reconfiguration scenario $Reconfiguration^{a}{}_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]}$, **Then** each other agent Ag_b ($b \in [1, n]\backslash\{a\}$) has to apply the scenario $Reconfiguration^{b}{}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]}$ (Fig. 3). We denote in the following by *idle agent* each agent Ag_b ($b \in [1, n]$) which is not required to apply any reconfiguration when others perform scenarios defined in *CM*. In this case:

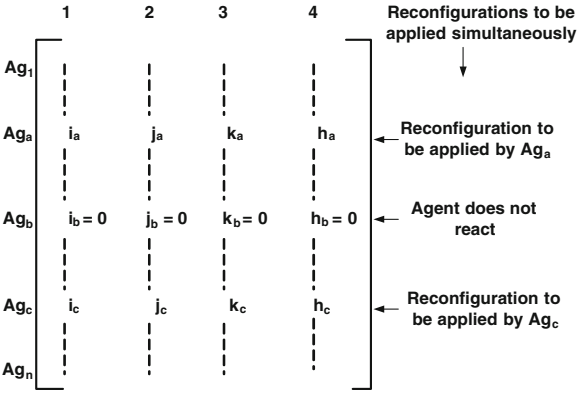


Fig. 3 The coordination matrix

$$CM[b,1] = CM[b,2] = CM[b,3] = CM[b,4] = 0$$
$$cond^{a}_{CM[a,1]} = cond^{a}_{CM[a,2]} = cond^{a}_{CM[a,3],CM[a,4]} = True$$

We propose a communication protocol between agents to manage concurrent distributed reconfiguration scenarios. We guarantee a coherent behavior of the whole distributed system by defining a *Coordination Agent* (denoted by $CA(\xi(Sys))$) which handles the Coordination Matrices of $\xi(Sys)$ to control the rest of agents (i.e. Ag_a, $a \in [1,n]$) as follows:

- When a particular agent Ag_a ($a \in [1,n]$) should apply a reconfiguration scenario $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ (i.e. under well-defined conditions), it sends the following request to $CA(\xi(Sys))$ to obtain its authorization:

$$request(Ag_a, CA(\xi(Sys)), Reconfiguration^{a}_{i_a,j_a,k_a,h_a}).$$

- When $CA(\xi(Sys))$ receives this request that corresponds to a particular coordination matrix $CM \in \xi(Sys)$ and if CM has the highest priority between all matrices of $Concur(CM) \cup \{CM\}$, then $CA(\xi(Sys))$ informs the agents that have simultaneously to react with Ag_a as defined in CM. The following information is sent from $CA(\xi(Sys))$:
  For each Ag_b, $b \in [1,n] \setminus \{a\}$ and $CM[b,i] \neq 0, \forall i \in [1,4]$: $reconfiguration(CA(\xi(Sys)), Ag_b, Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$
- According to well-defined conditions in the device of each Ag_b, the $CA(\xi(Sys))$ request can be accepted or refused by sending one of the following answers:
  - If $cond^{b}_{i_b} = cond^{b}_{j_b} = cond^{b}_{k_b,h_b} =$ True
    then the following reply is sent from Ag_b to $CA(\xi(Sys))$: $possible\_reconfig(Ag_b, CA(\xi(Sys)), Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$.
  - Else the following reply is sent from Ag_b to $CA(\xi(Sys))$: $not\_possible\_reconfig(Ag_b, CA(\xi(Sys)), Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$.

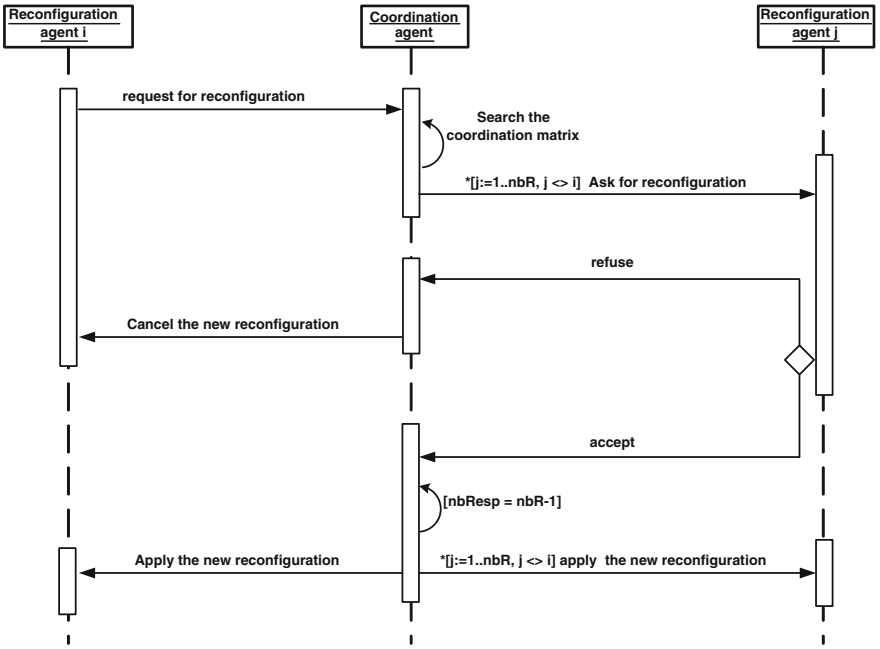


Fig. 4 The communication scenario

- If $CA(\xi(Sys))$ receives positive answers from all agents, then it authorizes reconfigurations in the concerned devices: For each Ag_b, $b \in [1, n]$ and $CM[b, i] \neq 0$, $\forall i \in [1, 4]$, *apply* $(Reconfiguration^{b}_{CM[b,1],CM[b,2],CM[b,3],CM[b,4]})$ in $device_b$. Else If $CA(\xi(Sys))$ receives a negative answer from a particular agent, then
  - If the reconfiguration scenario $Reconfiguration^{a}_{i_a,j_a,k_a,h_a}$ allows optimizations of the whole system behavior, then $CA(\xi(Sys))$ refuses the request of Ag_a by sending the following reply: $refused\_reconfiguration(CA(\xi(Sys)), Ag_a, Reconfiguration^{a}_{CM[a,1],CM[a,2],CM[a,3],CM[a,4]})$.

When a Reconfiguration Agent (denoted by RA_i) needs to apply a new reconfiguration, it sends a request to the Coordination Agent. The Coordination Agent asks all the known Reconfiguration Agents (denoted by RA_j, $\forall j \in [1..NbR], j <> i$ where NbR represents the number of Reconfiguration Agents) if it is possible to apply the new reconfiguration introduced as parameter. The Reconfiguration Agent (RA_j) studies this proposition and sends its response which may be accept or refuse the new reconfiguration (depending on its related state). Whenever the Coordination Agent receives positive responses from all the Reconfiguration Agents ($RA_j \forall j \in [1..NbR], j <> i$) (i.e. the number of positives answers is equal to $NbR-1$), then it decides to apply the new reconfiguration for all Reconfiguration Agents RA_j ($\forall j \in [1..NbR]$) by sending a confirmation message. Whenever the Coordination Agent receives only one negative response from a Reconfiguration Agents

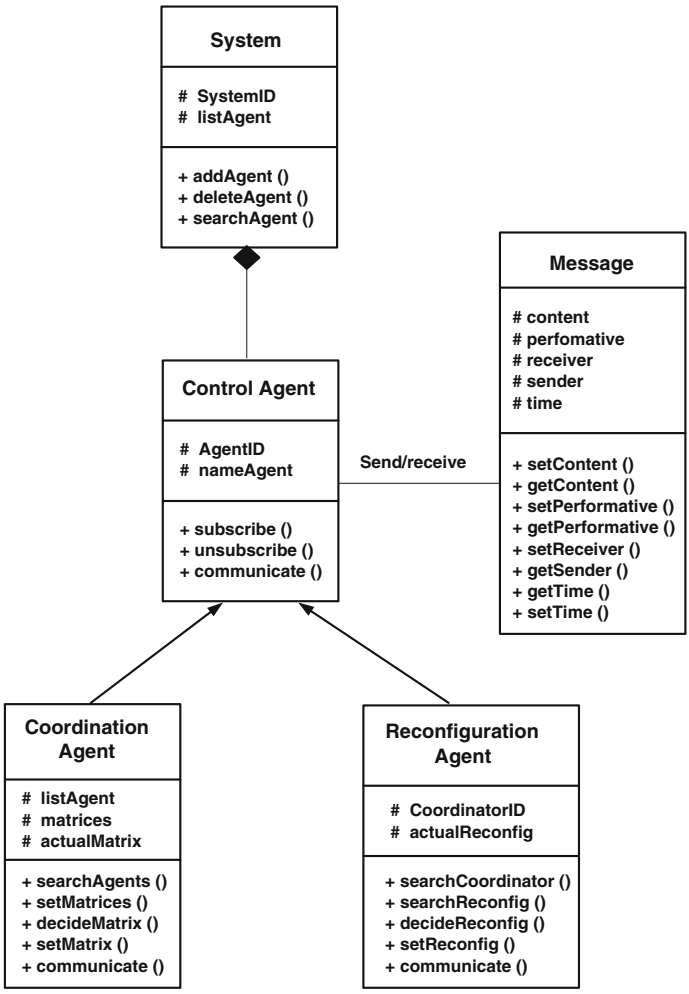


Fig. 5 The agent-based control in a distributed system

($RA_j, j \in [1..NbR], j <> i$), it decides to cancel this reconfiguration and informs the corresponding agent by its decision (i.e RA_i). Figure 4 depicts the interaction between Reconfiguration and Coordination agents to ensure dynamic reconfiguration in a distributed system.

Before sending or receiving a message, the Reconfiguration Agent searches the Coordination Agent with the method *searchCoordinator*(). The Coordination Agent in its turn searches also the list of Reconfiguration Agents with the method *searchAgents*().

The method *receive*() used by both Coordination Agent and Reconfiguration Agent permits to receive a message sent by another agent. Whenever *receive*() is

invoked through *CA_Communicate*() and *RA_Communicate*() methods, if the agent does not receive a message, it is blocked (but without blocking the other activities of the same agent).

A message is defined by the following data: (1) *content*: the subject of the message (such as the reconfiguration to be applied); (2) *performative*: the performative indicates what the sender wants to achieve (for example ACCEPT, REFUSE, CANCEL, CONFIRM); (3) *time*: it is necessary to treat messages ordered by time; (4) *sender*: the agent emitting the message; and (5) *receiver*: the agent receiving the message. Figure 5 depicts the different classes such as *ControlAgent*, *CoordinationAgent*, *ReconfigurationAgent*, *Message* and *System*.

In the following, we present the *Communicate* method defined for both Reconfiguration and Coordination Agent. The *CA_Communicate* method defined for the Coordination Agent has as variables: (1) *i* representing the reconfiguration agent which initiates the request of reconfiguration; (2) *j* which corresponds to the reconfiguration agent receiving the request of reconfiguration from the coordination agent; (3) *NbR* which represents the total number of reconfiguration agents; (4) *NbResp* considered as the current number of responses approving the new reconfiguration by the reconfiguration agents; (5) *matrix* representing the new matrix to be applied if all the reconfiguration agents accept.

Algorithm *CA_Communicate*()

```
begin
switch (step)
case 0:
// Wait a request from a Reconfiguration Agent
    reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = REQUEST)
            i ← reply.getSender();
            Matrix ← decideMatrix(reply.getContent());
            step++;
    else
        block();
    break;

    case 1:
// Send the proposition to all Reconfiguration Agents
    for j = 1 to NbR do
        if (j <> i)
            msg.addReceiver(reconfigurationAgents[j]);
            msg.setContent(Matrix[j]);
            msg.setPerformative(PROPOSE);
            msg.setTime(currentTime());
            send(msg);
    step++;
    break;
```

```
    case 2:
// Receive all accept/refusals from Reconfiguration Agents      reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = ACCEPT)
            nbResp++;
            if (nbResp = nbR-1)
                step++;
        else
            if (reply.getPerformative() = REFUSE)
                step ← 4;
    else
        block();
    break;

    case 3:
// Send accept response to all Reconfiguration Agents
    for j = 1 to NbR do
        msg.addReceiver(reconfigurationAgents[j]);
        msg.setPerformative(CONFIRM);
        msg.setTime(currentTime());
        msg.setContent(Matrix[j]);
        send(msg);
        setMatrix(Matrix);
    step ← 0;
    break;

    case 4:
// Send refuse response to the Reconfiguration Agent i
    msg.addReceiver(reconfigurationAgents[i]);
    msg.setPerformative(CANCEL);
    msg.setTime(currentTime());
    send(msg);
    step ← 0;
    break;
end
```

The *RA_Communicate* method defines the Reconfiguration Agent behavior as follows: (1) whenever the Reconfiguration Agent receives a request to apply a new reconfiguration by the Coordination Agent, it evaluates this proposition and decides whether to accept or to refuse it. The Reconfiguration Agent sends its response. (2) whenever the Reconfiguration Agent receives a confirmation to apply the new reconfiguration from the Coordination Agent, then it applies it.

Algorithm *RA_Communicate***()**

```
begin
switch (step)
case 0:
// Wait a request from a Coordination Agent
```

```
    reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = REQUEST)
            newReconfig ← reply.getContent();
            response.setReceiver(CoordinatorID);
            if (decideReconfig(newReconfig))
                response.setPerformative(ACCEPT);
            else
                response.setPerformative(REFUSE);
            send(response)
            step++;
    else
        block();
    break;

    case 1:
// Wait the response from a Coordination Agent
    reply ← receive();
    if (reply != null)
        if (reply.getPerformative() = CONFIRM)
            setReconfig(newReconfig);
        step ← 0;
    break;
end
```

We developed a complete tool "ProtocolReconf", to verify the communication protocol. The tool "ProtocolReconf" offers the possibility to create the Reconfiguration and Coordination Agents by introducing the necessary parameters. It is required to define the different scenarios that the Reconfiguration Agent can support so that when a modification occurs in the system, it should look for the convenient reconfiguration. For the Coordination Agent, it is necessary to define the set of Coordination Matrices to apply to the whole system [21].

5 Real-Time Task: Definition, Dynamic Reconfiguration and Scheduling

In this section, we present a Real-Time Task as a general concept independently from any real-time operating system, its dynamic reconfiguration, the scheduling between several tasks and the implementation in a specific real-time operating system (which is RTLinux).

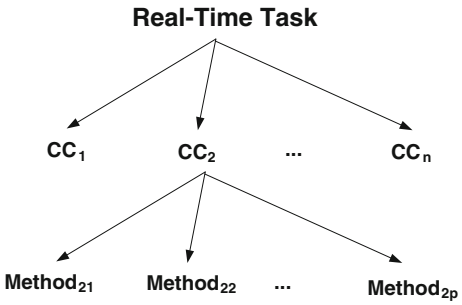


Fig. 6 Real time task

5.1 Real Time Task Definition

A real time task is considered as a process (or a thread depending on the Operating System) having its own data (such as registers, stack, ...) which is in competition with other tasks to have the processor execution. A task is handled by a Real-Time Operating System (RTOS) which is a system satisfying explicitly response-time constraints by supporting a scheduling method that guarantees response time especially to critical tasks.

In this paragraph, we aim to present a real-time task as a general concept independently from any real-time operating system.

To be independent from any Real-Time Operating System and to be related to our research work, we define a task τ_i as a sequence of Control Components, where a Control Component is ready when its preceding Control Component completes its execution. $\tau_{i,j}$ denotes the j-th Control Component of τ_i (Fig. 6). Thus, our application consists of a set of periodic tasks $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$. All the tasks are considered as periodic this is not a limitation since non-periodic task can be handled by introducing a periodic server.

Running Example. *In the FESTO Benchmark Production System, the tasks* τ_1 *to* τ_9 *execute the following functions:*

- (τ_1) *Feeder pushes out cylinder and moves backward/back;*
- (τ_2) *Converter pneumatic sucker moves right/left;*
- (τ_3) *Detection Module detects workpiece, height, color and material;*
- (τ_4) *Shift out cylinder moves backward/forward;*
- (τ_5) *Elevator elevating cylinder moves down/up;*
- (τ_6) *Rotating disc workpiece present in position and rotary indexing table has finished a 90 rotation;*
- (τ_7) *Driller 1 machine drills workpiece;*
- (τ_8) *Driller 2 machine drills workpiece;*
- (τ_9) *WarehouseCylinder removes piece from table.*

In the following paragraphs, we introduce the meta-model of a task. We study also the dynamic reconfiguration of tasks. After that, we introduce the task scheduling.

Finally, we present the task implementation within RTLinux as a Real-Time Operating System.

5.2 A Meta-model Task

In this chapter, we extend the work presented in [42] by studying both a task and a scheduler in a general real-time operating system where each task is characterized by:

identifer: each task τ_i has a name and an identifier.

temporal properties: each task τ_i is described by a deadline D_i (which corresponds to the maximal delay allowed between the release and the completion of any instance of the task), a period T_i, a worst-case execution time C_i. It is released every T_i seconds and must be able to consume at most C_i seconds of CPU time before reaching its deadline D_i seconds after release ($C_i \leq D_i \leq T_i$). We assume that these attributes are known, and given as constants (Table 2).

constraints: resources specification ρ_i, precedence constraints and/or QoS properties to be verified.

state: A Real-Time Operating System implements a finite state machine for each task and ensures its transition. The state of a task may be in one of the following possible states Ready, Running, Blocked or Terminated. Every task is in one of a few different states at any given time:

Ready The task is ready to run but waits for allocation of the processor. The scheduler decides which ready task will be executed next based on priority criterion (i.e. the task having the highest priority will be assigned to the processor).

Blocked A task cannot continue execution because it has to wait (there are many reasons such that waiting for event, waiting on semaphore or a simple delay).

Running In the running state, the processor is assigned to the task, so that its instructions can be executed. Only one task can be in this state at any time, while all the other tasks can be simultaneously in other states.

Terminated When a task terminates its execution, the task allocator deletes it and releases the resources taken by this task (Fig. 7).

priority: each task is assigned a priority value which may be used in the scheduling.

$$\tau_i = (D_i; C_i; T_i; I_i; O_i; \rho_i; (CC_i^1, \ldots, CC_i^{n_i}));$$

- a deadline D_i;
- an execution time C_i;
- a period T_i;
- a set of inputs I_i;
- a set of outputs O_i;

Table 2 A task set example

Task	Comp. time C_i	Period T_i	Deadline D_i
τ_1	20	70	50
τ_2	20	80	80
τ_3	35	200	100
τ_4	62	90	81

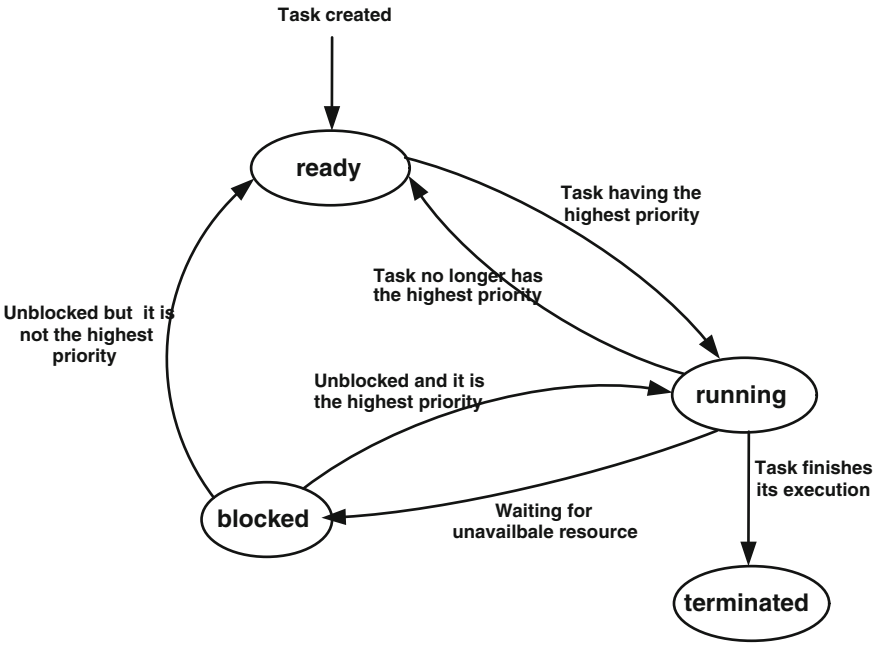


Fig. 7 Task states

- a set of constraints ρ_i;
- a set of n_i Control Components ($n_i \geq 1$) such that the task τ_i is constituted by $CC_i^1, CC_i^2, \ldots, CC_i^{n_i}$.

One of the core components of an RTOS is the task scheduler which aims to determine which of the ready tasks should be executing. If there are no ready tasks at a given time, then no task can be executed, and the system remains idle until a task becomes ready (Fig. 8).

Running Example. *In the FESTO Benchmark Production System, when the task* τ_1 *is created, it is automatically marked as Ready task. At the instant* $t1$*,it is executed by the processor (i.e. it is in the Running state). When the task* τ_1 *needs a resource at the instant* $t2$*, it becomes blocked. Whenever the resource is available at the*

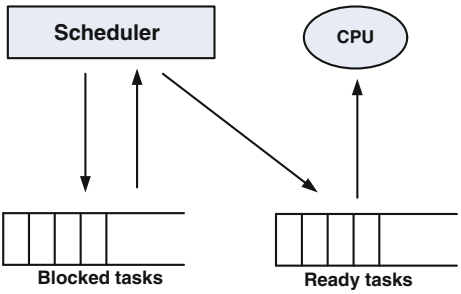


Fig. 8 Scheduling task

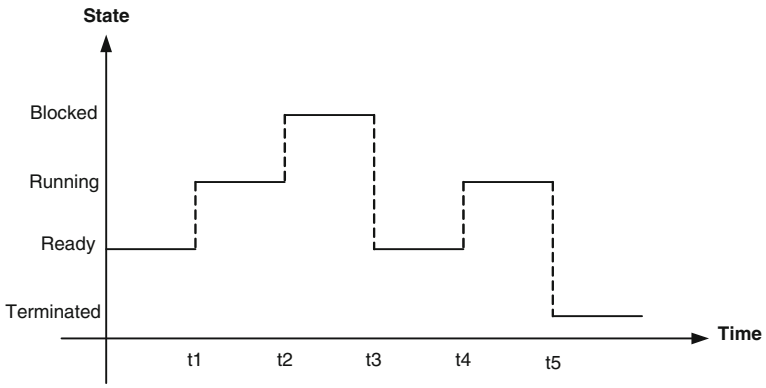


Fig. 9 The variation of states related to the task τ_1

instant $t3$, *the task* τ_1 *is transformed into ready state. Finally, it is executed again since the time* $t4$. *It is terminated at the instant* $t5$ (Fig. 9).

A scheduler related to a real-time operating system is characterized by (Fig. 10):

readyTask: a queue maintaining the set of tasks in ready state.
executingTask: a queue maintaining the set of tasks in executing state.
minPriority: the minimum priority assigned to a task.
maxPriority: the maximum priority assigned to a task.
timeSlice: the threshold of preempting a task (the quantity of time assigned to a task before its preemption).

Several tasks may be in the ready or blocked states. The system therefore maintains a queue of blocked tasks and another queue for ready tasks. The latter is maintained in a priority order, keeping the task with the highest priority at the top of the list.When a task that has been in the ready state is allocated the processor, it makes a state transition from ready state to running state. This assignment of the processor is called dispatching and it is executed by the dispatcher which is a part of the scheduler.

Running Example. *In the FESTO Benchmark Production System, we consider three tasks* τ_1, τ_2 *and* τ_3. *having as priority* $p1$, $p2$ *and* $p3$ *such that* $p1 < p2 < p3$.

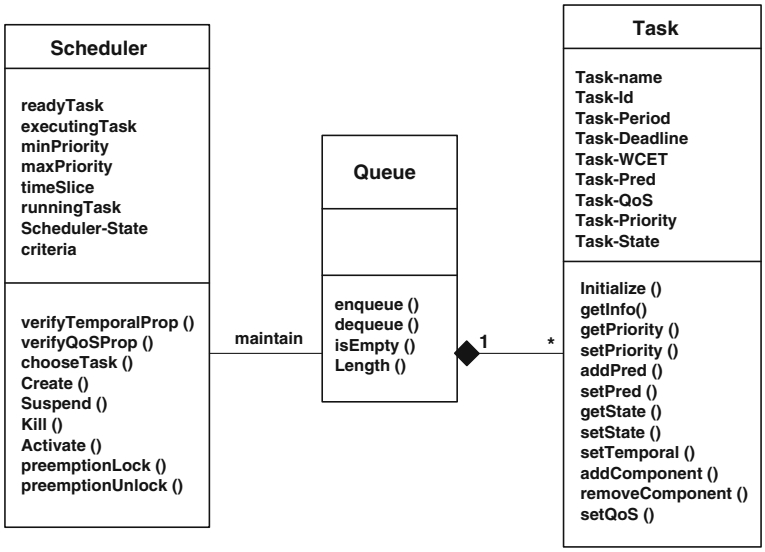


Fig. 10 The real time operating system

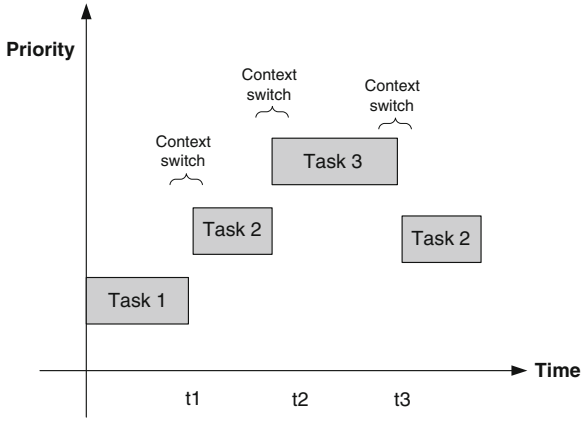


Fig. 11 The context switch between tasks

We suppose that the task τ_1 *is running when the task* τ_2 *is created at the instant t1. As a consequence, there is a context switch so that the task* τ_1 *stays in a ready state and the other task* τ_2 *begins its execution as it has higher priority. At the instant t*2*, the task* τ_3 *which was already blocked waiting a resource, gets the resource. As the task* τ_3 *is the highest priority, the task* τ_2 *turns into ready state and* τ_3 *executes its routine. The task* τ_3 *continues processing until it has completed, the scheduler enables* τ_2 *to become running* (Fig. 11).

5.3 Feasible and Safety Dynamic Reconfiguration of Tasks

We want to study the system's safety during reconfiguration scenarios. In fact, we want to keep tasks running while dynamically reconfiguring them. We assume for such system's task several software processes which provide functional services, and assume also reconfiguration processes that apply required modifications such as adapting connections, data or internal behaviors of the component. The execution of these different tasks is usually critical and can lead to incorrect behaviors of the whole system. In this case, we should schedule which process should be firstly activated to avoid any conflict between processes. Consequently, we propose in this section to synchronize processes for coherent dynamic reconfigurations applied to several tasks.

5.3.1 Reconfiguration and Service Processes

We want in this section to synchronize service and reconfiguration processes of a task according to the following constraints: (1) whenever a reconfiguration process is running, any new service process must wait until its termination; (2) a reconfiguration process must wait until the termination of all service processes before it begins its execution; (3) it is not possible to execute many reconfiguration processes in parallel; (4) several service processes can be executed at the same time. To do that, we use semaphores and also the famous synchronization algorithm between readers and writer processes such that executing a service plays the role of a reader process and reconfiguring a task plays the role of a writer process. In the following algorithm, we define *serv* and *reconfig* as semaphores to be initialized to 1. The shared variable *Nb* represents the number of current service processes associated to a specific task. Before the execution of a service related to a task, the service process increments the number *Nb* (which represents the number of service processes). It tests if it is the first process (i.e. *Nb* is equal to one). In this case, the operation P(reconfig) ensures that it is not possible to begin the execution if there is a reconfiguration process.

```
P(serv)
Nb ← NB + 1
 if (NB = 1) then
   P(reconfig)
 end if
V(serv)
```

After the execution of a service related to a task, the corresponding process decrements the number *Nb* and tests if there is no service process (i.e. *Nb* is equal to zero). In this case, the operation *V*(*reconfig*) authorizes the execution of a reconfiguration process.

```
P(serv)
Nb ← NB − 1
```

```
    if (NB = 0) then
      V(reconfig)
    end if
  V(serv)
```

Consequently, each service process related to a task does the following instructions:

Algorithm 2: execute a service related to a task

```
begin service
P(serv)
    Nb ← NB + 1
    if (NB = 1) then
      P(reconfig)
    end if
V(serv)

    execute the service

    P(serv)
    Nb ← NB - 1
    if (NB = 0) then
      V(reconfig)
    end if
V(serv)
end service
```

Running Example. *Let us take as a running example the task Test related to the EnAS system. To test a piece before elevating it, this component permits to launch the Test Service Process*. Figure 12 *displays the interaction between the objects Test Service Process,Service semaphore and Reconfiguration semaphore. The flow of events from the point of view of Test Service Process is the following:* (1) *the operation P*(*serv*) *leads to enter in critical section for Service semaphore;* (2) *the number of services is incremented by one;* (3) *if it is the first service, then the operation P*(*reconfig*) *permits to enter in critical section for Reconfiguration semaphore;* (4) *the operation V*(*serv*) *leads to exit from critical section for Service semaphore;* (5) *the Test Service Process executes the corresponding service;* (6) *before modifying the number of service, the operation P*(*serv*) *leads to enter in critical section for Service semaphore;* (7) *the number of services is decremented by one;* (8) *if there is no service processes, then the operation V*(*reconfig*) *permits to exit from critical section for Reconfiguration semaphore;* (9) *the operation V*(*serv*) *leads to liberate Service semaphore from its critical section.*

With the operation *P*(*reconfig*), a reconfiguration process verifies that there is no reconfiguration processes nor service processes which are running at the same time. After that, the reconfiguration process executes the necessary steps and runs the operation *V*(*reconfig*) in order to push other processes to begin their execution. Each reconfiguration process specific to a task realizes the following instructions:

Algorithm 3: reconfigure a task

begin reconfiguration
P(reconfig)
    execute the reconfiguration
V(reconfig)
end reconfiguration

Running Example. *Let us take as example the task Elevate related to EnAS system. The agent needs to reconfigure this task which permits to launch the Elevate Reconfiguration Process*. Figure 13 *displays the interaction between the following objects Elevate Reconfiguration Process and Reconfiguration semaphore. The flow of events from the point of view of Elevate Reconfiguration Process is the following:* (1) *the operation P*(*reconfig*) *leads to enter in critical section for Reconfigura-*

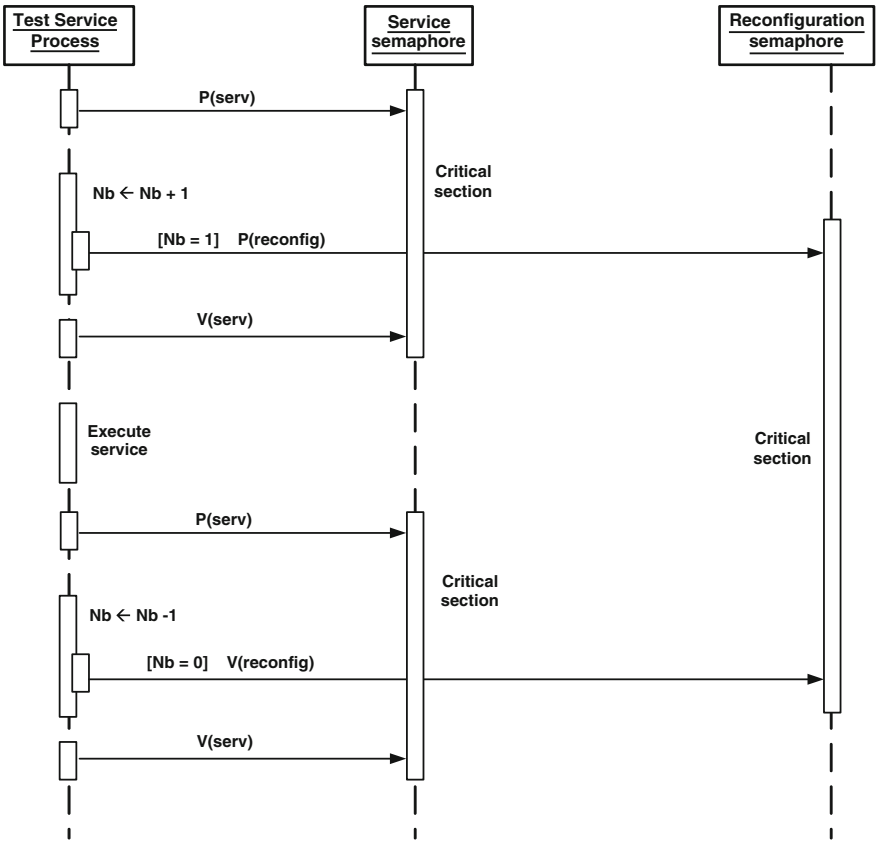


Fig. 12 The service process scenario

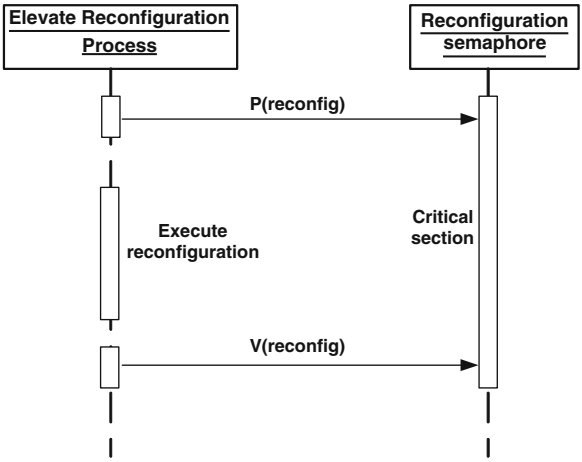


Fig. 13 The reconfiguration process scenario

tion semaphore; (2) *the Elevate Reconfiguration Process executes the corresponding reconfiguration;* (3) *the operation V*(*reconfig*) *leads to liberate Reconfiguration semaphore from its critical section.*

5.3.2 Verification of Safety of the Synchronization

To verify the safety of the synchronization, we should verify if the different constraints mentioned above are respected.

First property: whenever a reconfiguration process is running, any service processes must wait until the termination of the reconfiguration. Let us suppose that there is a reconfiguration process (so the integer *reconfig* is equal to zero and the number of current services is zero). When a service related to this component is called, the number of current services is incremented (i.e. it is equal to 1) therefore the operation $P(reconfig)$ leads the process to be in a blocked state (as the integer *reconfig* is equal to zero). When the reconfiguration process terminates the reconfiguration, the operation $V(reconfig)$ permits to liberate the first process waiting in the semaphore queue. In conclusion, this property is validated.

Second property: whenever a service process is running, any reconfiguration processes must wait until the termination of the service. Let us suppose that there is a service process related to a component (so the number of services is greater or equal to one which means that the operation $P(reconfig)$ is executed and *reconfig* is equal to zero). When a reconfiguration is applied, the operation $P(reconfig)$ leads this process to be in a blocked state (as the *reconfig* is equal to zero). Whenever the number of service processes becomes equal to zero, the operation $V(reconfig)$ allows to liberate the first reconfiguration process waiting in the semaphore queue. As a conclusion, this property is verified.

Third property: whenever a reconfiguration process is running, it is not possible to apply a new reconfiguration process until the termination of the first one. Let us suppose that a reconfiguration process is running (so *reconfig* is equal to zero). Whenever, a new reconfiguration process tries to execute, the operation $P(reconfig)$ puts it into a waiting state. After the reconfiguration process which is running is terminated, the operation $V(reconfig)$ allows to liberate the first reconfiguration waiting process. Consequently, this property is respected.

Fourth property: whenever a service process is running, it is possible to apply another process service. Let us suppose that a service process $P1$ is running. Whenever, a new service process $P2$ tries to begin the execution, the state of $P2$ (activated or blocked) depends basically on the process $P1$:

- if $P1$ is testing the shared data *Nb*, then the operation $P(serv)$ by the process $P2$ leads it to a blocking state. When the process $P1$ terminates the test of the shared data *Nb*, the operation $V(serv)$ allows to launch the process waiting in the semaphore's queue.
- if $P1$ is executing its service, then the operation $P(serv)$ by the process $P2$ allows to execute normally.

Thus, this property is validated.

5.4 Task Scheduling with Priority Ceiling Protocol

How to schedule periodic tasks with precedence and mutual exclusion constraints is considered as important as how to represent a task in a general real-time operating system. In our context, we choose the priority-driven preemptive scheduling used in the most real-time operating systems. The semaphore solution can lead to the problem of priority inversion which consists that a high priority task can be blocked by a lower priority task. To avoid such problem, we propose to apply the priority inheritance protocol proposed by Sha et al. [49].

The priority inheritance protocol can be used to schedule a set of periodic tasks having exclusive access to common resources protected by semaphores. To do so, each semaphore is assigned a priority ceiling which is equal to the highest priority task using this semaphore. A task τ_i is allowed to enter its critical section only if its assigned priority is higher than the priority ceilings of all semaphores currently locked by tasks other than τ_i.

Schedulability test for the priority ceiling protocol: a set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following inequalities hold, $\forall i$, $1 \leq i \leq n$,

$$C_1/T_1 + C_2/T_2 + \cdots + C_i/T_i + B_i/T_i \leq i(2^{1/i} - 1)$$

where B_i denotes the worst-case blocking time of a task τ_i by lower priority tasks.

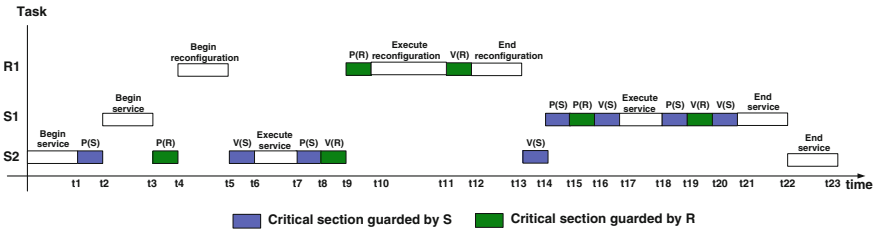


Fig. 14 The priority ceiling protocol applied to three tasks R1, S1 and S2

Table 3 The event and its corresponding action in Fig. 14

Event	Action
t0	S2 begins execution
t1	S2 locks S. The task S2 inherits the priority of S
t2	The task S1 is created. As it has more priority than S2, it begins its execution
t3	The task S1 fails to lock S as its priority is not higher than the priority ceiling of the locked S. The task S2 resumes the execution with the inherited priority of S
t4	The task S2 locks R. The task S2 inherits the priority of R. The task R1 is created and preempts the execution of S2 as it has the highest priority
t5	The task R1 fails to lock R as its priority is not higher than the priority ceiling of the locked R. The task S2 resumes the execution of the critical section
t6	The task S2 unlocks S
t7	The task S2 executes a service
t8	The task S2 locks S
t9	The task S2 unlocks R and therefore has as priority the same as S. The task R1 becomes having the highest priority. As it has more priority than S2, it resumes its execution
t10	The task R1 locks R
t11	The task R1 executes the reconfiguration
t12	The task R1 unlocks R
t13	The task R1 terminates its execution
t14	The task S2 unlocks S (thus S2 becomes having the lowest priority). Therefore, the task S1 resumes its execution
t15	The task S1 locks S
t16	The task S1 locks R
t17	The task S1 unlocks S
t18	The task S1 executes its service
t19	The task S1 locks S
t20	The task S1 unlocks R
t21	The task S1 unlocks S
t22	The task S1 achieves its execution
t23	The task S2 resumes the execution and terminates its service

Running Example. *In the FESTO Benchmark Production System, we consider three tasks* $R1$ *(a reconfiguration task),* $S1$ *and* $S2$ *(service tasks) having as priority*

$p1, p2$ and $p3$ such that $p1 > p2 > p3$. The sequence of processing steps for each task is as defined in the section previous paragraph where S (resp. R) denotes the service (resp. reconfiguration) semaphore:

$$R1 = \{\ldots P(R) \textit{ execute reconfiguration } V(R)\ldots\}$$
$$S1 = \{\ldots P(S)\ldots P(R)\ldots V(S) \textit{ execute service } P(S)\ldots V(R)\ldots V(S)\ldots\}$$
$$S2 = \{\ldots P(S)\ldots P(R)\ldots V(S) \textit{ execute service } P(S)\ldots V(R)\ldots V(S)\ldots\}$$

Therefore, the priority ceiling of the semaphore R is equal to the task R1 (because the semaphore R is used by the tasks R1, S1 and S2 and we know that the task R1 is the highest priority) and the priority ceiling of the semaphore S is equal to the task S1 (because the semaphore S is used by the tasks S1 and S2 and the priority task of S1 is higher). We suppose that the task $S2$ is running when the task $S1$ is created at the instant t3. We suppose also that the task $R1$ is created at the instant t5. Fig. 14*, a line in a high level indicates that the task is executing, a line in a low level indicates that the the task is blocked or preempted by another task.* Table 3 *explains more in details the example.*

6 Conclusion

This chapter deals with Safety Reconfigurable Embedded Control Systems. We propose conceptual models for the whole component-based architecture. We define a multi-agent architecture where a Reconfiguration Agent is affected to each device of the execution environment to handle local automatic reconfigurations, and a Coordination Agent is defined to guarantee safe distributed reconfigurations. To deploy a Control Component in a Real-Time Operating System, we define the concept of real-time task in general (especially its characteristics). The dynamic reconfiguration of tasks is ensured through a synchronization between service and reconfiguration processes to be applied. We propose to use the semaphore concept for this synchronization such that we consider service processes as readers and reconfiguration processes as writers. We propose to use the priority ceiling protocol as a method to ensure the scheduling between periodic tasks with precedence and mutual exclusion constraints. The main contributions presented through this work are: the study of Safety Reconfigurable Embedded Control Systems from the functional to the operational level and the definition of a real-time task independently from any real-time operating system as well as the scheduling of these real-time tasks considered as periodic tasks with precedence and mutual exclusion constraints. The chapter's contribution is applied to two Benchmark Production Systems FESTO and EnAS available at Martin Luther University in Germany.

References

1. M. Akerholm, J. Fredriksson, *A Sample of Component Technologies for Embedded Systems*
2. Y. Al-Safi, V. Vyatkin, *An ontology-based reconfiguration agent for intelligent mechatronic systems* (Springer, New York, 2007). Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems
3. C. Angelov, K. Sierszecki, N. Marian, Design models for reusable and reconfigurable state machines, in *EUC 2005, LNCS 3824* eds. by L.T. Yang et al., International Federation for Information Processing (2005), pp. 152–163
4. C. Bidan, V. Issarny, T. Saridakis, A. Zarras, A dynamic reconfiguration service for CORBA, in *CDS 98: Proceedings of the International Conference on Configurable Distributed Systems* (IEEE Computer Society, 1998)
5. K.-J. Cho, et al., A study on the classified model and the agent collaboration model for network configuration fault management. Knowl. Based Syst., 177–190 (2003)
6. M. Colnaric, D. Verber, W.A. Halang, A data-centric approach to composing embedded, real-time software components. J. Syst. Softw. **74**, 25–34 (2005)
7. F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, *Scheduling in Real-Time Systems* (Wiley, New York, 2002)
8. I. Crnkovic, *Component-based Approach for Embedded Systems* (2003)
9. I. Crnkovic, M. Larsson, *Building Reliable Component-based Software Systems* (Artech House, 2002)
10. I. Crnkovic, M. Larsson, *Grid Information Services for Distributed Resource Sharing* (Artech House, UK, 2002). Building reliable component-based software systems
11. M. de Jonge, Developing Product Lines with Third-Party Components. Electronic Notes in Theoretical Computer Science (2009), pp. 63–80
12. EN50126, *Railway Applications the Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS)* (Comite Europeen de Nomalisation Electrotechnique, 1999)
13. EN50128, *Railway Applications Software for Railway Control and Protection Systems* (Comite Europeen de Nomalisation Electrotechnique, 2002)
14. EN50129, *Railway Applications Safety Related Electronic Systems for Signalling* (Comite Europeen de Nomalisation Electrotechnique, 2002)
15. EN954, *Safety of Machinery Safety-related Parts of Control Systems* (Comite Europeen de Nomalisation Electrotechnique, 1996)
16. R. Faller, Project experience with IEC 61508 and its consequences. Saf. Sci. **42**, 405–422 (2004)
17. T. Genler, O. Nierstrasz, B. Schonhage, *Components for Embedded Software The PECOS Approach*
18. T. Genssler, et al., *PECOS in a Nutshell* (2002)
19. A. Gharbi, H. Gharsellaoui, M. Khalgui, S. Ben Ahmed, Functional safety of distributed embedded control systems, in *Handbook of Research on Industrial Informatics and Manufacturing Intelligence: Innovations and Solutions*, eds. by M.A. Khan, A.Q. Ansari (2011)
20. A. Gharbi, M. Khalgui, S. Ben Ahmed, Functional safety of discrete event systems. First Workshop of Discrete Event Systems (2011)
21. A. Gharbi, M. Khalgui, S. Ben Ahmed, Inter-agents communication protocol for distributed reconfigurable control software components. The International Conference on Ambient Systems Networks and Technologies (ANT), 8–10 Nov 2010
22. A. Gharbi, M. Khalgui, S. Ben Ahmed, Model checking optimization of safe control embedded components with refinement. 5th International conference on Design and Technology of Integrated Systems in Nanoscale Era (2010)
23. A. Gharbi, M. Khalgui, S. Ben Ahmed, Optimal model checking of safe control embedded software components. 15th IEEE International Conference on Emerging Technologies and Factory Automation (2010)

24. A. Gharbi, M. Khalgui, H.M. Hanisch, Functional safety of component-based embedded control systems. 2nd IFAC Workshop on Dependable Control of Discrete Systems (2009)
25. http://www.program-Transformation.org/Tools/KoalaCompiler. Last accessed on 11 July 2010
26. IEC 1131–3, *Programmable Controllers, Part 3: Programming Languages* (International Electrotechnical Commission, Geneva, 1992)
27. IEC 61508, *Functional Safety of Electrical/Electronic Programmable Electronic Systems: Generic Aspects*. Part 1: General requirements (International Electrotechnical Commission, Geneva, 1992)
28. IEC60880, *Software for Computers in the Safety Systems of Nuclear Power Stations* (International Electrotechnical Commission, 1987)
29. IEC61511, *Functional Safety: Safety Instrumented Systems for the Process Industry Sector* (International Electrotechnical Commission, Geneva, 2003)
30. IEC61513, *Nuclear Power Plants Instrumentation and Control for Systems Important to Safety General Requirements for Systems* (International Electrotechnical Commission, Geneva, 2002)
31. G. Jiroveanu, R.K. Boel, A distributed approach for fault detection and diagnosis based on Time Petri Nets. Math. Comput. Simul., 287–313 (2006)
32. M. Kalech, M. Linder, G.A. Kaminka, Matrix-based representation for coordination fault detection: a formal approach. Comput. Vis. Image Underst.
33. A. Ketfi, N. Belkhatir, P.Y. Cunin, *Automatic Adaptation of Component-based Software Issues and Experiences* (2002)
34. M. Khalgui, H.M. Hanisch, A. Gharbi, Model-checking for the functional safety of control component-based heterogeneous embedded systems. 14th IEEE International conference on Emerging Technology and Factory Automation (2009)
35. J. Kramer, J. Magee, The evolving Philosophers problem: dynamic change management. IEEE Trans. Softw. Eng. **16** (1990)
36. P. Leitao, Agent-based distributed manufacturing control: A state-of-the-art survey. Eng. Appl. Artif. Intell. (2008)
37. A.J. Massa, *Embedded Software Development with eCos*, 1st edn (Prentice Hall, Upper Saddle River, NJ, USA, 2002)
38. S. Merchant, K. Dedhia, *Performance Comparison of RTOS* (2001)
39. C. Muench, *The Windows CE Technology Tutorial: Windows Powered Solutions for the Developer* (Addison Wesley, Reading, 2000)
40. S. Olsen, J. Wang, A. Ramirez-Serrano, R.W. Brennan, Contingencies-based reconfiguration of distributed factory automation. Robot. Comput. Integr. Manuf., 379–390 (2005) (Safety Reconfigurable Embedded Control Systems 31)
41. G.A. Papadopoulos, F. Arbab, *Configuration and Dynamic Reconfiguration of Components Using the Coordination Paradigm* (2000)
42. P. Pedreiras, L. Almeida, *Task Management for Soft Real-Time Applications based on General Purpose Operating, System* (2007)
43. G. Pratl, D. Dietrich, G. Hancke, W. Penzhorn, A new model for autonomous, networked control systems. IEEE Trans. Ind. Inform. **3**(1) (2007)
44. QNX Neutrino, *Real Time Operating System User Manual Guide* (2007)
45. A. Rasche, A. Polze, *ReDAC—Dynamic Reconfiguration of distributed component-based applications with cyclic dependencies* (2008)
46. A. Rasche, W. Schult, Dynamic updates of graphical components in the .NET Framework, in *Proceedings of SAKS07 Workshop* eds. by A. Gharbi, M. Khalgui, M.A. Khan, vol. 30 (2007)
47. M.N. Rooker, C. Sunder, T. Strasser, A. Zoitl, O. Hummer, G. Ebenhofer, *Zero Downtime Reconfiguration of Distributed Automation Systems : The eCEDAC Approach* (Springer, New York, 2007). Third International Conference on Industrial Applications of Holonic and Multi-Agent Systems
48. G. Satheesh Kumar, T. Nagarajan, Experimental investigations on the contour generation of a reconfigurable Stewart platform. IJIMR **1**(4), 87–99 (2011)
49. L. Sha, R. Rajkumar, J.P. Lehoczky, Priority inheritence protocols: an approach to real-time synchronization. IEEE Trans. Comput. **39**(9), 1175–1185 (1990)

50. D.D. Souza, A.C. Wills, *Objects, Components and Frameworks: The Catalysis Approach* (Addison-Wesley, Reading, MA, 1998)
51. D.B. Stewart, R.A. Volpe, P.K. Khosla, Design of dynamically reconfigurable real-time software using port-based objects. IEEE Trans. Softw. Eng. **23**, 592–600 (1997)
52. C. Szyperski, D. Gruntz, S. Murer, *Component Software Beyond Object- Oriented Programming* (The Addison-Wesley Component Software Series, 2002)
53. R. van Ommering, F. van der Linden, J. Kramer, J. Magee, *The Koala Component Model for Consumer Electronics Software* (IEEE Computer, Germany, 2000), pp. 78–85
54. M. Winter, *Components for Embedded Software—The PECOS Approach*
55. R. Wuyts, S. Ducasse, O. Nierstrasz, A data-centric approach to composing embedded, real-time software components. J. Syst. Softw. (74), 25–34 (2005)

Low Power Techniques for Embedded FPGA Processors

Jagrit Kathuria, Mohammad Ayoub Khan, Ajith Abraham and Ashraf Darwish

Abstract The low-power techniques are essential part of VLSI design due to continuing increase in clock frequency and complexity of chip. The synchronous circuit operates at highest clock frequency. These circuits drive a large load because it has to reach many sequential elements throughout the chip. Thus, clock signals have been a great source of power dissipation because of high frequency and load. Since, clock signals are used for synchronization, they does not carry any information and certainly doesn't perform any computation. Therefore, disabling the clock signal in inactive portions of the circuit is a useful approach for power dissipation reduction. So, by using clock gating we can save power by reducing unnecessary clock activities inside the gated module. In this chapter, we will review some of the techniques available for clock gating. The chapter also presents Register-Transfer Level(RTL) model in Verilog language. Along with RTL model we have also analyzed the behaviors of clock gating technique using waveform.

J. Kathuria (✉)
HMR Institute of Technology and Management, New Delhi, India
e-mail: jagritkathuria@gmail.com

M. A. Khan
Department of Computer Science and Engineering, Sharda University, Gr. Noida, India
e-mail: ayoub@ieee.org

A. Abraham
Machine Intelligence Research Labs (MIR Labs), Auburn, Washington, USA
e-mail: ajith.abraham@ieee.org

A. Darwish
Helwan University, Cairo, Egypt
e-mail: amodarwish@yahoo.com

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_11, © Springer-Verlag Berlin Heidelberg 2014

1 Introduction

The Moore's law states that the density of transistors on an Integrated Circuit (IC) will double approximately every two years. However, there are many challenges. The power density of the IC increases exponentially in every generation of technology. We also know that bipolar and nMOS transistors consume energy even in a stable combinatorial state. However, CMOS transistors consume lower power largely because power is dissipated only when they switch states, and not when the state is steady. The power consumption has been always an important area of research in circuit design. Also, there is a paradigm shift from traditional single core computing to multi-core System-on-Chip (SoC). A SoC consists of computational intellectual property cores, analog components, interface and ICs to implement a system on a single-chip. More than billion transistors are expected to be integrated on a single-chip. Multiple cores can run multiple instructions simultaneously, increasing overall speed for programs amenable to parallel computing. Processors were originally developed with only one core. The traditional multiprocessor techniques are no longer efficient because of issues like congestion in supplying instructions and data to the many processors. The Tilera processors has a switch in each core to route data through an on-chip mesh network to avoid data congestion [12]. Hence, SoCs with hundreds of IP cores are becoming a reality. The growth of number of cores in single-chip is shown in Fig. 1. The fundamental idea to reduce the power consumption is to disconnect the clock when the device is idle. This technique is called clock gating. In synchronous circuits the clock signal is responsible for significant part of power dissipation (up to 40 %) [3]. The power density has once again increased in multi-core processing and SoC. The Register Transfer Level (RTL) clock gating is the most common technique used for optimization and improving efficiency but still it leaves one question that how efficiently the gating circuitry has been designed. The gated clock is widely accepted technique in order to optimize power that can be applied at system, gate and RTL level. The clock gating can save more power by shutting off the clock to register if there is no change in the state. The clock gating technique has ability to retain the state of register while clock is shut off.

1.1 Clock Gating

The clock signal and associated circuitry dominates in every synchronous design. The clock signal switches every cycle, thus it has an activity factor of 1. Therefore, the clock circuitry consumes huge amount of the on-chip dynamic power. To reduce the power clock gating technique has been widely used to limit the activity factor. Thus, clock gating reduces the dynamic power dissipation by disconnecting the clock source from an unused or ideal circuit block. We also understand that

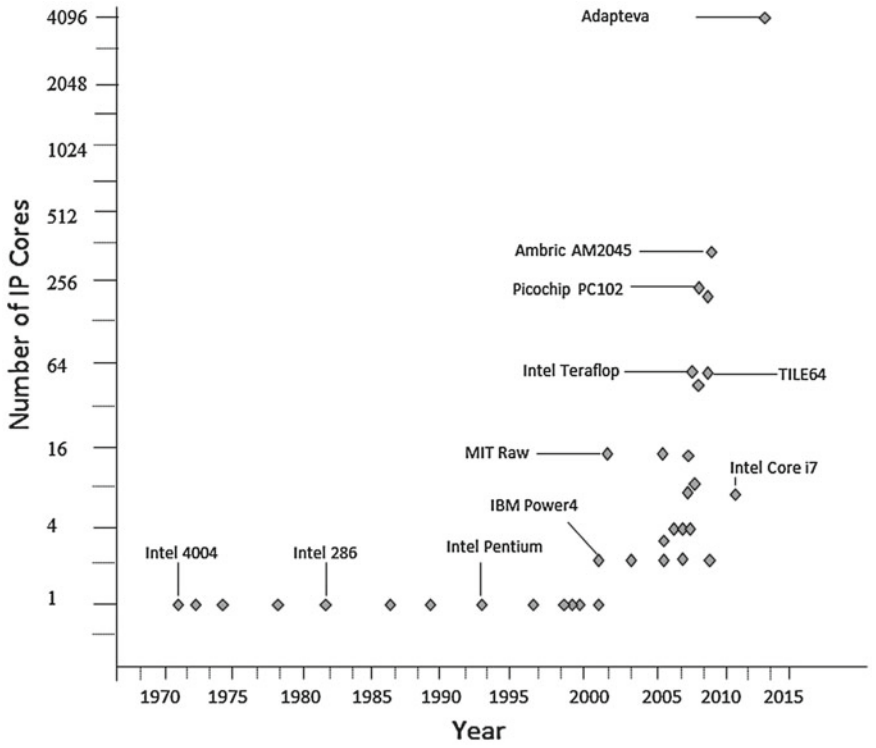


Fig. 1 Growth of IP cores in single-chip

power is directly proportional to voltage and the frequency of the clock as shown in the following equation:

$$Power = Capacitance \times (Voltage)^2 \times (Frequency) \tag{1}$$

Generally, the system clock is connected to the clock pin on every flip-flop in the design. Therefore, we can observe main sources of power consumption as follows:

1. *Power consumption in combinational logic due to evaluation on each clock cycle.*
2. *Power consumed by flip-flops in case of steady inputs.*

The clock gating can reduce power consumed by flip-flops and combinational network. The simplest approach for clock gating is to identify a set of flip-flops who shares a common enable signal. Generally, the enable signal is ANDed with the clock to generate the gated clock. The gated clock is then fed to the clock ports of all of the flip-flops. The clock gating is a good technique to reduce the power, however, several considerations in implementing clock gating is needed. We have listed some of the important consideration as follows:

1. *The enable signal shall remain stable during high period of the clock. The enable signal can switch when clock is in low period.*
2. *The glitches at the gated clock should be avoided.*
3. *The clock skew at gating circuitry must be avoided. Hence, gating technique need a careful design.*

This chapter presents an exhaustive survey and discussion on several techniques for clock gating. The chapter also presents analysis on RTL design and simulation. Also, chapter discusses some of the fundamental mechanisms employed for power reduction.

2 Timing Analysis

In the steady state behavior of combinational circuit the output is a function of the inputs under the assumption that inputs have been stable for a long time relative to the delays in the circuit. However, the actual delay from an input change to output change in a real circuit is non-zero which depends on many factors.
Glitches—The unequal arrival of input signals produces transient behavior in a logic circuit that may differ from what is predicted by a steady-state analysis. As shown in Fig. 2, the output of a circuit may produce a short pulse, often called glitch; at a time steady state analysis predicts that the output should not change [14].
Hazards—A hazard is a circuit which may produce a glitch. A hazard exists when a circuit has some possibility of producing glitches [6, 14]. This is an unwanted glitch that occurs due to unequal path or unequal propagation delays through a combinational circuit.There are two types of hazards as follows.

1. Static Hazard
   (a) Static-1 Hazard—If the output momentarily goes to sate '0' when the output is expected to remain in state '1' as per the steady state analysis, the hazard of this nature is known as static-1 Hazard.
   (b) Static-0 Hazard—If the output momentarily goes to sate '1' when the output is expected to remain in state '0' as per the steady state analysis, the hazard of this nature is known as static-0 Hazard.
2. Dynamic Hazard—When the output is supposed to change from 0 to 1 (or from 1 to 0), the circuit may go through three or more transients and produce more than one glitch. Such multiple glitch situations are known as dynamic hazards.

In the Fig. 3, we can see to glitch when *en* is switching from high to low and *CLK* is switching from low to high. In a similar fashion we can see a glitch in Fig. 4 where *en* and *CLK* both are switching from high to low (Fig. 5).

```
Verilog code for AND based gating:
module DFF (input d,en,clk,reset, output reg q);
    and a1(gclk,clk,en);
    always @(posedge gclk or negedge reset)
    begin
        if (!reset)
            q<=1'b0;
```

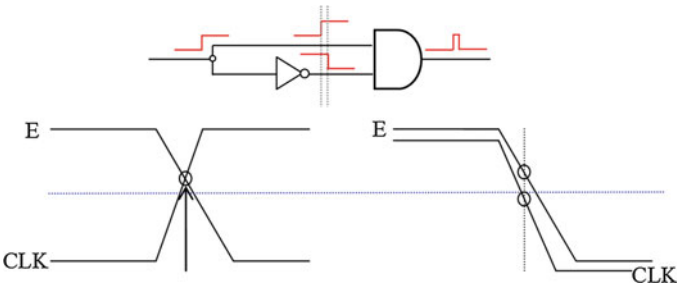


Fig. 2 Glitches when *En* and *CLK* is applied randomly at the inputs of AND based clock gating

2.1 Clock Gating Techniques

Most of the clock-gating has been applied at RTL level. In this section we present six different techniques for clock gating at RTL level. The RTL clock-gating can be applied at three level:

1. System-level
2. Sequential
3. Combinational

The system-level clock-gating stops the clock for an entire circuitry. The system-level technique effectively disables entire functionality of the system. While as combinational and sequential selectively suspend clocking while the block continues to produce an output. This chapter considers a counter for evaluating various clock gating technique. We start our discussion with AND gate as fundamental clock gating technique (Fig. 8).

2.2 AND Gate

In the beginning, many authors have suggested use of AND gate for clock gating due to simple logic design [4, 14, 15]. Here, we will analyze the response of the sequential circuit while applying fundamental AND gate technique for clock gating. In order to control the state of clock we need an enable (*En*) input to AND gate other

```
Verilog code for AND based gating:
module DFF (input d,en,clk,reset, output reg q);
    and a1(gclk,clk,en);
    always @(posedge gclk or negedge reset)
    begin
        if (!reset)
            q<=1'b0;
        else
            q<=d;
        end
  endmodule
```

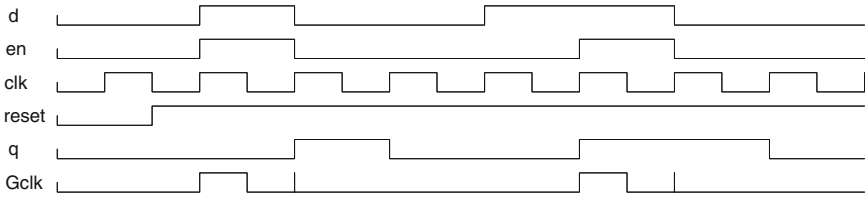


Fig. 3 Glitches when *En* and *CLK* is applied randomly at the inputs of AND based gating

than clock (*CLK*). To demonstrate the concept we present schematic, RTL code and output waveform. Throughout the chapter we have took 4-bit counter to apply clock gating technique. Let's first analyze basic 4 bit nagative edge counter as shown in Fig. 6. As shown in Fig. 7, initially at *reset* = 0, the counter initialized to 0 and thereafter when *reset* = 1 the counter increments at each negative edge of the clock.

Let's Analyze response of counter circuit with AND clock gating as shown in Fig. 8. In Fig. 9 we can observe that enable is stable to high when *clk* is rising, therefore, counter is incremented on active edge of the clock. However, in Fig. 10, when en is starts changing from positive edge to the next positive edge, then counter increments one extra time, due to tiny *glitch*. Another scenario of negative edged triggered is shown in Fig. 11. We can observe response of counter when *en* changes from clock cycle starting from negative edge to the next negative edge. In this case output of the counter changes after one clock cycle. Therefore, counter works correctly as the inputs are supplied for sufficient amount of time and signal is stable as show in Fig. 11.

However, if the input timing of *en* and *Clk* are not synchronized then it may lead to unpredictable results. In Fig. 12, any momentary changes in *en* signal while *Clk* is active will produce hazard in *Gclk*. This situation may be dangerous and could jeopardize the correct functioning of the entire system [6].

```
Verilog code for NOR based gating:
module DFF (input d,en,clk,reset, output reg q);
    nor a1(gclk,clk,~en);
    always @(posedge gclk or negedge reset)
    begin
        if (!reset)
            q<=1'b0;
        else
            q<=d;
        end
endmodule
```

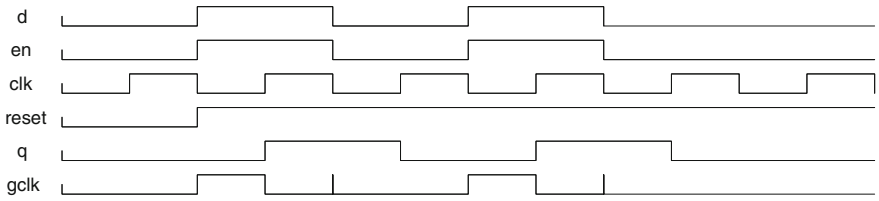


Fig. 4 Glitches when *En* and *CLK* is applied randomly at the inputs of NOR based Gating

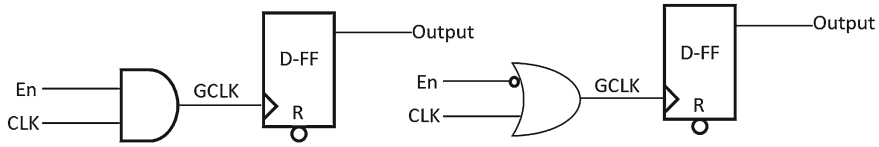
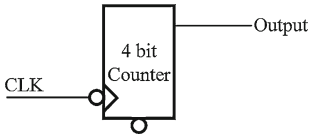


Fig. 5 Schematic of basic AND and NOR based gating technique

Fig. 6 Basic counter(negative edge triggered)



2.3 NOR Gate

Using NOR gate for clock gating is a very suitable technique for reduce the power where we need actions to be performed on positive edge of the global clock [6, 14]. Therefore, we will now observe response of counter based on NOR gating. The schematic of NOR based clock gating is shown in Fig. 13. In this figure we can observe that counter will work when *en* signal is *active* because we have connected an inverter at the input of the NOR gate. In Fig. 14 waveform shows incorrect output of the counter when *en* changes to *1*, from negative edge of the clock to the next negative edge of the clock. In this case the counter is positive edge

triggered so by observing *GClk* we can say that due to small glitch when *en* signal changes to *inactive* at negative edge of the clock the counter increments once more. The important thing is that we can use this configuration where we want to analyze circuit response on positive edge of the clock. However, in this case the target system should be negative edge triggered with NOR Clock gating, we can verify this from Fig. 16. In the Fig. 15 correct output of the counter is shown when counter is positive edge triggered in this case output is correct because *en* signal is changing from positive edge of the clock to the next positive edge of the clock. In the Fig. 17, we have shown a major problem of hazards when any momentary hazard at the enable could be pass on to the Gclk when clk = '0' this situation is particularly very dangerous and could jeopardize the correct functioning of the entire system [6].

2.4 Latch Based and Clock Gating

Till now, we have analyzed two type of clock gating technique based on AND and NOR gate. Now, we will discuss latch based AND clock gating technique as shown

```
Verilog code for basic counter:
module counter(input reset,clk, output reg [4:1]q);
always @(negedge clk,negedge reset)
begin
    if(!reset)
        q<=4'b0000;
    else
        q<=q+1;
end
endmodule
```

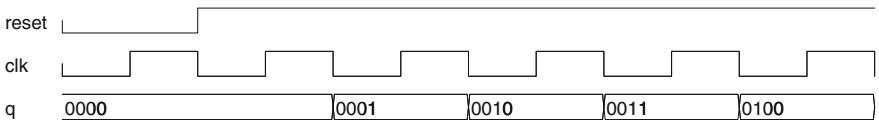


Fig. 7 Correct output of the basic counter without clock gating

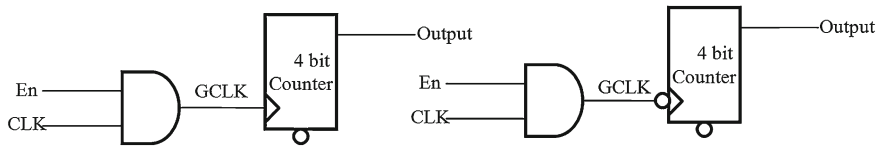


Fig. 8 Clock gating using AND gate circuit

```
Verilog code for positive edged triggered counter:
module gated (input reset,clk,en, output reg[4:1] q);
    and no1(Gclk,en,clk);
    always @(posedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

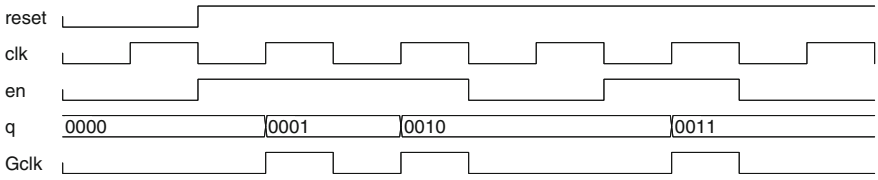


Fig. 9 Correct output of positive edge triggered counter

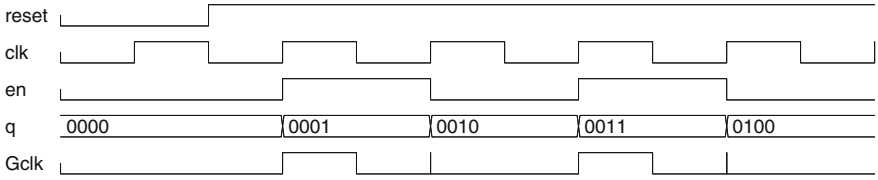


Fig. 10 Output with glitch in positive edge triggered counter

in Fig. 18. In this design we have inserted a latch between one input of AND gate and *En* input signal. In the previous designs, *En* was directly connected to the AND gate input, but here this will come through a latch. The latch is needed for correct behavior, because *En* may have have hazards that must not propagate through AND gate when global clock *GLCK* is *high* [1, 11, 13]. Moreover, we can notice that the delay of the logic for the computation of *En* may on the critical path of the circuit, therefore, effect must be taken into account during time verification [1, 5, 8, 11]. We can observe from Fig. 19 that counter will take one extra clock cycle to change the state and after that it will work normally until *En* is de-asserted. Also, at the time of de-assertion it will take one extra clock cycle to change the state.

In Fig. 20 we have verified that unwanted outputs due to hazards at the *En* has been eliminated. In Fig. 21, we can observe that when controlling latch is positive and counter is also positive edge triggered then output of the counter is incorrect because it increments the one more time even when *En* is *low* due to a tiny glitch due (Fig. 22).

```
Verilog code for negative edge triggered counter:
module gated (input reset,clk,en, output reg[4:1] q);
    and no1(Gclk,en,clk);
    always @(negedge Gclk,negedge reset)
    begin
    if(!reset)
        q<=4'b0000;
    else
        q<=q+1;
    end
endmodule
```

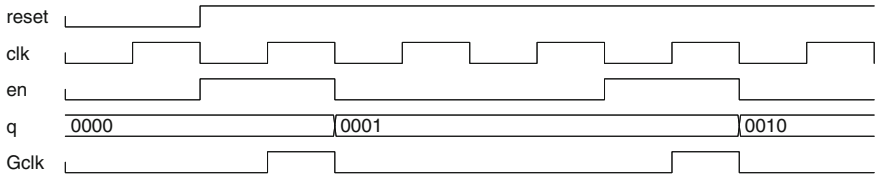


Fig. 11 Correct output of negative edge triggered counter

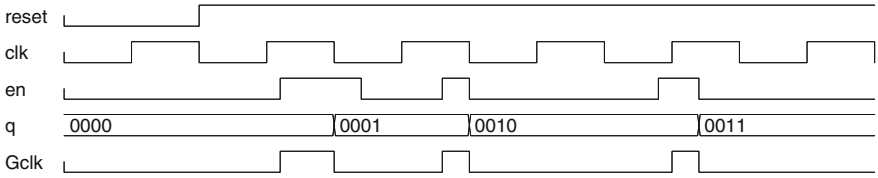


Fig. 12 Hazards problem when AND clock gating circuitry is used

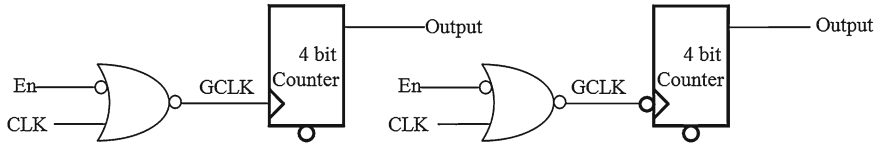


Fig. 13 Clock gating using NOR gate circuit

2.5 Latch Based NOR Clock Gating

Latch based NOR gated technique is shown in Fig. 23. As we can observe from figure that we have inserted a latch between one input of NOR gate and *En* input signal. In the NOR based clock gating the *En* signal was directly connected to NOR gate input, but in this design *En* is coming through a latch.

We can observe from Fig. 24 that initially counter is taking one extra clock cycle to change the state. Thereafter, this will work normal until *En* is de-asserted. At the

```
Verilog code for figure 14 and 15:
module gated (input reset,clk,en, output reg[4:1] q);
    nor no1(Gclk,~en,clk);
    always @(posedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

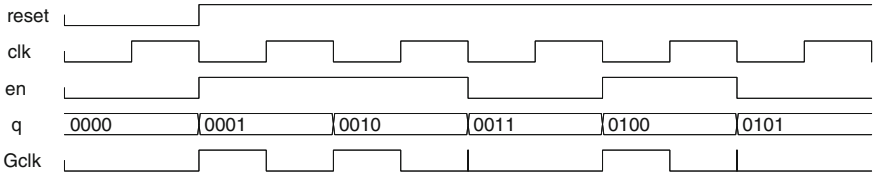


Fig. 14 Incorrect output of counter when counter is positive edge triggered

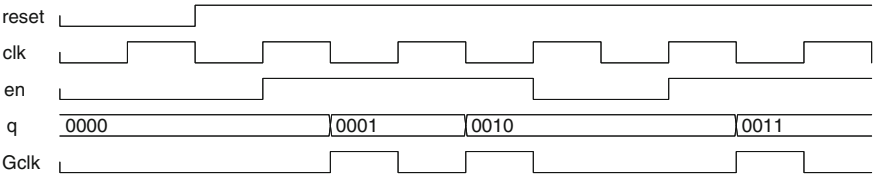


Fig. 15 Correct output of counter when counter is positive edge triggered

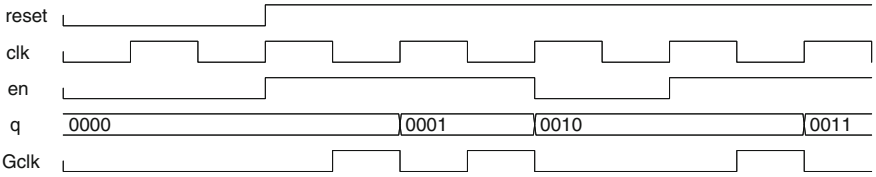


Fig. 16 Output of counter when enable changes from positive edge to next positive edge but counter is negative edge triggered

time of de-assertion also it will take one extra clock cycle to change the state. In Fig. 25, we have verified that unwanted outputs due to hazards at the *En* signal has been eliminated.

In Fig. 26, we can observe that when controlling latch is negative and counter is also negative edge triggered then output of the counter is incorrect because it

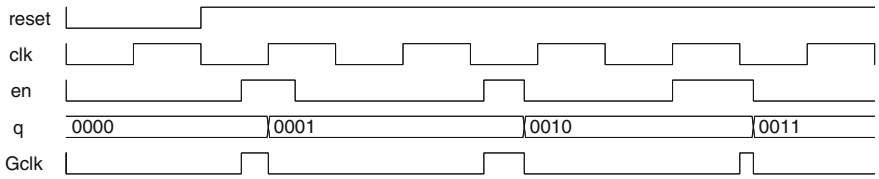


Fig. 17 Hazards problem when NOR gate is used for clock gating

increments the counter one extra time even when *En* is *low* due to a tiny glitch as shown in Fig. 27.

2.6 Multiplexer Based Clock Gating

In multiplexer based clock gating, we use multiplexer to close and open a feedback loop around a basic D-type flip-flop under the control of *En* signal as shown in Fig. 28. Therefore, the resulting circuit is simple, robust, and compliant with the rules of synchronous design. However, this approach takes fairly expensive because per bit one multiplexer is needed which is energy inefficient. This is because of switching at the clock input of a disabled flip-flop that amounts to wasting energy in discharging and recharging the associated node capacitances. In Figs. 29 and 30, we have shown the negative and positive edge triggered counter respectively. We can observe from these waveforms that when *En* is *high* then at each negative and positive edge of the clock respectively counter increments and when *En* goes *low* then counter holds the state.

3 New Design

In this section, we will discuss another efficient design that will save more power. In this circuit a clock gating cell for gating is used that is similar to latch based clock gating. The gated clock generation circuit is shown in Figs. 31 and 34 using negative latch and positive latch respectively. These circuits also have one comparison logic, first logic circuit and second logic circuit. This circuit saves power in such a way that even when target's device clock is ON, the controlling device's clock is OFF and also when the target device's clock is OFF then also controlling device's clock is OFF. This way we can save more power by avoiding unnecessary switching at clock signal [9]. When *En* becomes *high* at that time *GEN* is *low* so XNOR will produce $x =$'0' which goes to the first clock generation logic that generates clock for controlling device (latch). In first logic we have an OR gate which have Global Clock *GCLK* as an input at the other input of OR gate. This logic will generate a clock pulse that will drive the controlling latch when x turns to *low* (Figs. 32 and 33).

```
Verilog code for positive latch:
module dlatchP (input data,clk,reset,output reg q);
always @(clk, data)
  begin
if (reset == 1'b0)
q <= 1'b0;
else  if(clk==1'b1)
q <= data;
  end
endmodule
```

```
Verilog code for negative latch:
module dlatchN (input data,clk,reset,output reg q);
always @(clk, data)
  begin
if (reset == 1'b0)
q <= 1'b0;
else  if(clk==1'b0)
q <= data;
  end
endmodule
```

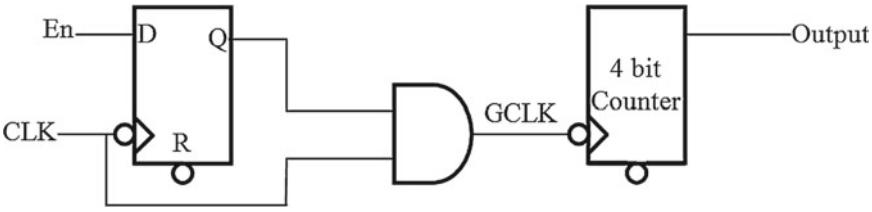


Fig. 18 Clock gating of negative edge counter using negative latch based AND gate circuit

The second clock generation logic has an AND gate with *GEN* and Global clock *GCLK* at its input. In the next clock pulse, when *GEN* turns to *high* the second clock generation logic which is an AND gate with *GEN* and Global clock *GCLK* at its input and when Gen goes *high* then it generates clock pulse that goes to the target device. Since *GEN* is *high* the XNOR will produce $x =$ '1' thus OR will produce at *CClk* constant *high* until *En* turns to *low*. This way *GClk* will be running and *CClk* will be at constant *high* state that means latch will hold its state without any switching. The circuit shown in Fig. 34 performs similar sequence of operations as explained for the circuit shown in Fig. 31. When *En* turns to *high* at that time *GEN* is *low* so XOR will produce $x =$'1' that goes to the first clock generation logic that

```
Verilog code for figure 19 and 20:
module gated (input reset,clk,en, output reg[4:1] q);
dlatchN dff(en,clk,reset,q1);
and no1(Gclk,q1,clk);
always @(negedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
        end
endmodule
```

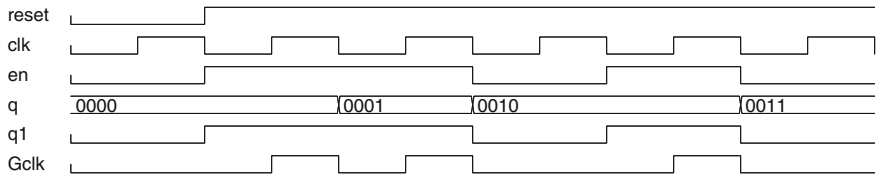


Fig. 19 Normal output of negative edge nounter when negative latch based AND gated clock is used

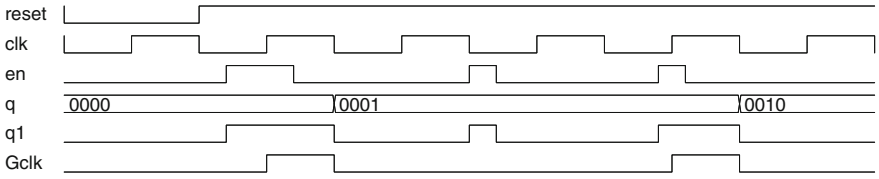


Fig. 20 Output of negative edge counter when there are some random Hazards at *En*

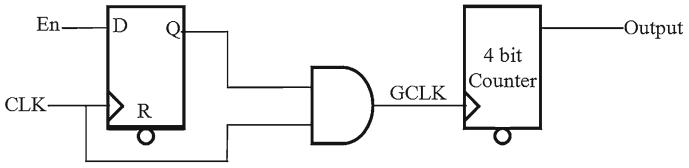


Fig. 21 Clock gating of positive edge counter using positive latch based AND gate circuit

generates clock for controlling device (Latch). In first logic we have an AND gate, which have global clock as input at the other input of AND gate. This logic will generate a clock pulse that will drive the controlling latch when *x* turns to *high*. In the next clock pulse, when *GEN* turns to *high* our second clock generation logic which is an OR gate which has Q* and Global clock at its input and when Q* goes

```
Verilog Code for figure 22:
module gated (input reset,clk,en, output reg[4:1] q);
dlatchP dff(en,clk,reset,q1);
and no1(Gclk,q1,clk);
always @(posedge Gclk,negedge reset)
    begin
    if(!reset)
        q<=4'b0000;
    else
        q<=q+1;
    end
endmodule
```

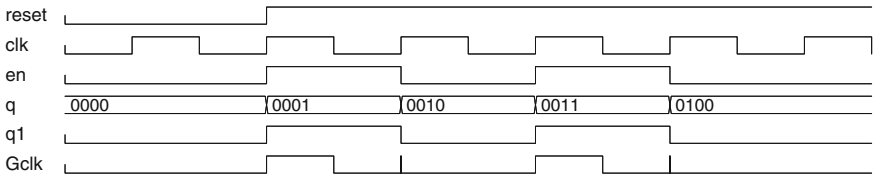


Fig. 22 Output of counter when latch is positive and counter is also positive edge triggered

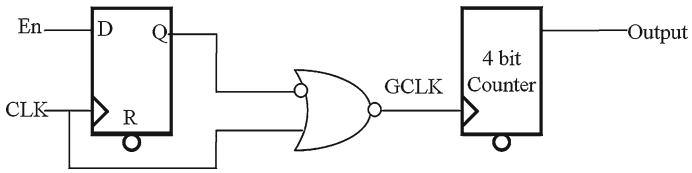


Fig. 23 Clock gating of negative edge counter using positive latch based NOR gate circuit

'0' it generates clock pulse that goes to the target device. Since *GEN* is *high* then XOR will produce $x = 0$ and OR will produce at *CClk* constant *low* until *En* turns to *low*. This way *GClk* will be running and *CClk* will be at constant *low* state that means latch will hold its state without any switching.

The output of counter for circuit shown in Fig. 31 is shown in Figs. 32 and 33. In Figs. 32 and 33 the enable signal changes from negative edge to next negative edge and positive edge to next positive edge respectively. We can also observe that target circuit is negative edge triggered and positive edge triggered respectively. The presented design produces correct output that gives us solution of the problem that persists in previous four types of clock gating. The output of counter for circuit shown in Fig. 34 is shown in Figs. 35 and 36. In Figs. 35 and 36 the enable signal changes from negative edge to next negative edge and positive edge to next positive edge respectively. The target circuitry is negative edge triggered and positive edge

```
Verilog code for figure 24 and 25:
module gated (input reset,clk,en, output reg [4:1]q);
dlatchP dff(en,clk,reset,q1);
    not no(n1,q1);
    nor no1(Gclk,n1,clk);
    always @(negedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

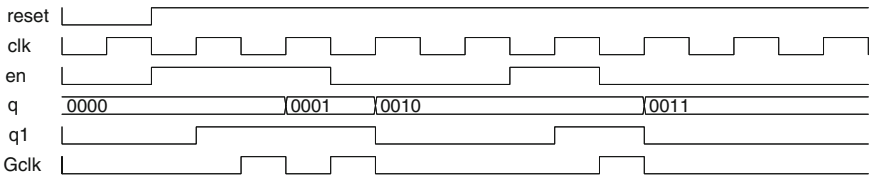


Fig. 24 Normal output of negative edge counter when positive latch based OR gated clock is used

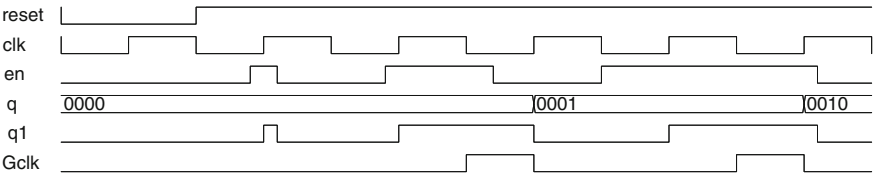


Fig. 25 Output of negative edge counter when there are some random Hazards at *En*

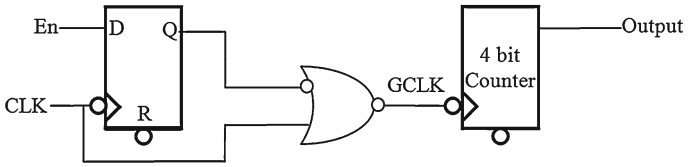


Fig. 26 Clock gating of negative edge counter using negative latch based NOR gate circuit

triggered respectively. The presented design produces correct output which gives us solution of the problem that persists in the previous four types of clock gating.

```
    dlatchN dff(en,clk,reset,q1);
    not no(n1,q1);
    nor no1(Gclk,n1,clk);
    always @(negedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

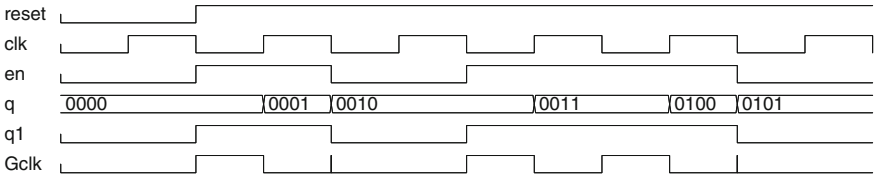


Fig. 27 Output of counter when latch is negative and counter also negative edge triggered

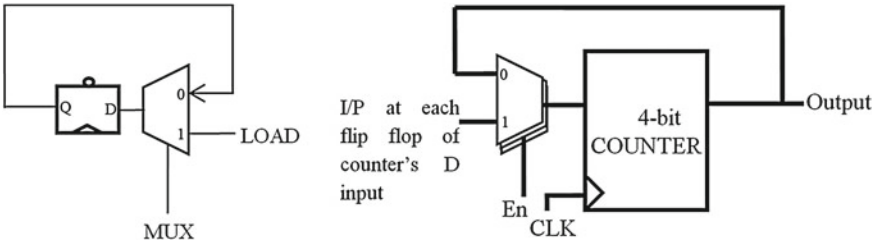


Fig. 28 Mux based clock gating for counter [5, 7, 8]

4 Conclusions

In this chapter, we have presented review of existing clock gating techniques. We have also discussed merits an demerits of these techniques. We have seen that first two techniques (AND and NOR based clock gating) have problem glitches if inputs are random and not stable for sufficient amount of time at inputs. While, in latch based AND and NOR techniques the problem of hazard has been removed, however glitches still exists. In the multiplexer based clock gating technique we don't have these problem, however, it consumes more area and power. The last design is more appropriate that removed glitches and hazards. In this design the clock gating applied at controllers side and also at target circuitry. This techniques saves more power than any other existing technique.

```
Verilog code for figure 29:
module Mux_counter(input en, reset, clk,
output reg [4:1]q);
    always @(negedge clk,negedge reset)
    begin
        if (!reset)
            q<=4'b0000;
        else
            if(en==1)
                q<=q+1;
    end
endmodule
```

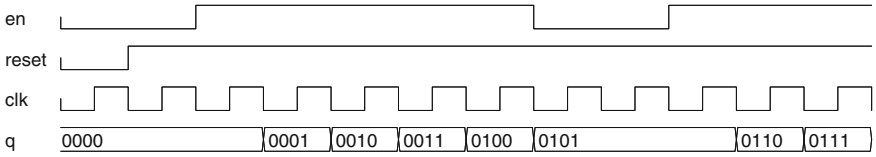


Fig. 29 Output of negative edge triggered counter with multiplexer based clock gating

```
Verilog Code for figure 30:
module Mux_counter(input en,reset,clk,
output reg [4:1]q);
always @(posedge clk,negedge reset)
begin
if (!reset)
q=4'b0000;
else
if(en==1)
q=q+1;
end
endmodule
```

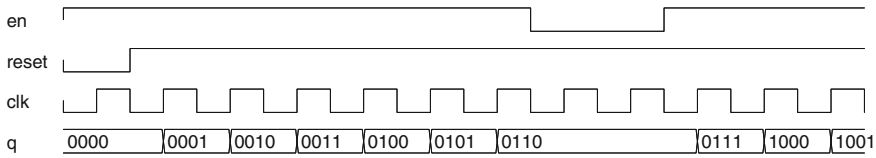


Fig. 30 Output of positive edge triggered counter with multiplexer based clock gating

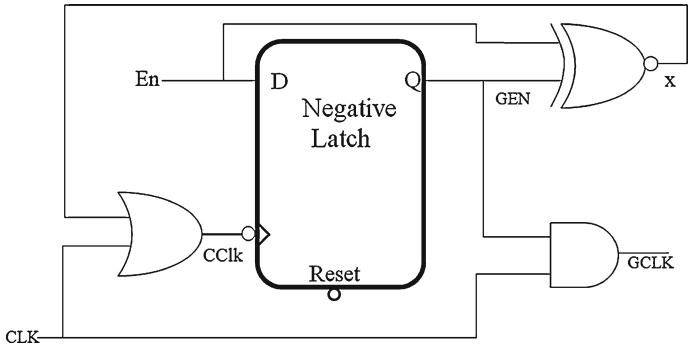


Fig. 31 Generation of gated clock when negative latch is used [9]

```
Verilog code for figure 32:
module gated(input en,reset,clk, output reg [4:1]q);
    or o1(CClk,x,clk);
    dlatchN d1(en,clk,reset,Gen);
    xnor xn(x,Gen,en);
    and a1(Gclk,Gen,clk);
    always @(negedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

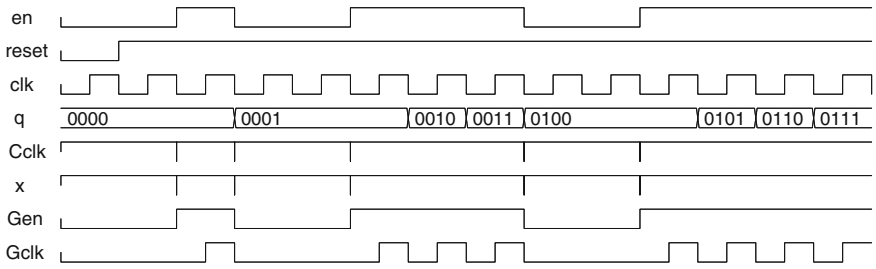


Fig. 32 Output of negative edge counter with gated clock for circuit shown in Fig. 31

```
Verilog code for figure 33:
module gated(input en,reset,clk, output reg [4:1]q);
    or o1(CClk,x,clk);
    dlatchN d1(en,clk,reset,Gen);
    xnor xn(x,Gen,en);
    and a1(Gclk,Gen,clk);
    always @(posedge Gclk,negedge reset)
    begin
    if(!reset)
        q<=4'b0000;
    else
        q<=q+1;
    end
endmodule
```

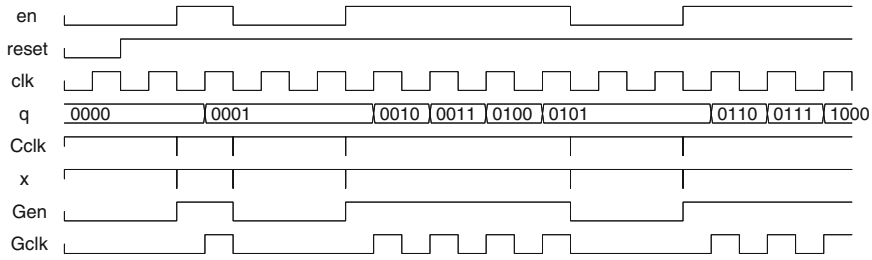


Fig. 33 Output of positive edge counter with gated clock for circuit shown in Fig. 31

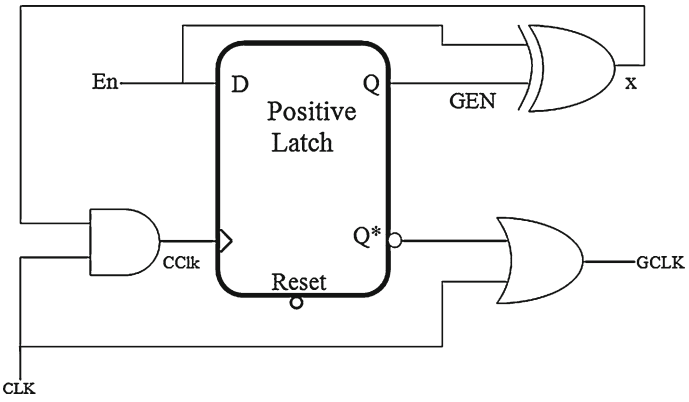


Fig. 34 Generation of gated clock when positive latch is used [9]

```
Verilog code for figure 35:
module gated(input en,reset,clk, output reg [4:1]q);
    and o1(CClk,x,clk);
    dlatchP d1(en,clk,reset,Gen);
    xor xn(x,Gen,en);
    or a1(Gclk,~Gen,clk);
    always @(negedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
        end
endmodule
```

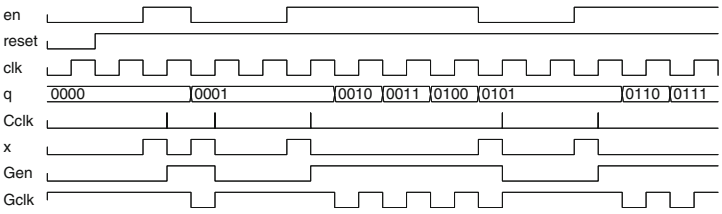


Fig. 35 Output of negative edge counter with gated clock for circuit shown in Fig. 34

```
Verilog code for figure 36:
module gated(input en,reset,clk, output reg [4:1]q);
    and o1(CClk,x,clk);
    dlatchP d1(en,clk,reset,Gen);
    xor xn(x,Gen,en);
    or a1(Gclk,~Gen,clk);
    always @(posedge Gclk,negedge reset)
    begin
        if(!reset)
            q<=4'b0000;
        else
            q<=q+1;
    end
endmodule
```

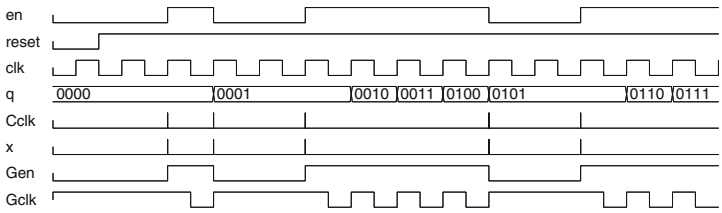


Fig. 36 Output of positive edge counter with gated clock for circuit shown in Fig. 34

References

1. L. Benini, G. De Micheli, E. Macii, M. Poncino, R. Scarsi, *Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers* (Transactions on Design Automation of Electronic Systems, Oct, 1999)
2. M. Dale, *Utilizing clock-gating efficiency to reduce power* (EE Times, India, 2008)
3. D. Dobberpuhl, R. Witek, A 200 MHz 64 b dual-issue CMOS microprocessor, IEEE International Solid-State Circuits Conference (1992), pp. 106–107
4. F. Emnett, M. Biegel, *Power reduction through RTL clock gating*. SNUG San Jose 2000
5. S. Huda, M. Mallick, J.H. Anderson, Clock gating architectures for FPGA power reduction. Field Programmable Logic and Applications (FPL) 2009
6. H. Kaeslin, *ETH Zurich digital integrated circuit design from VLSI architectures to CMOS fabrication* (Cambridge University Press, Cambridge, 2008)
7. T. Kitahara, F. Minami, T. Ueda, K. Usami, S. Nishio, M. Murakata, T. Mitsuhashi, A clock-gating method for low-power LSI design, TOSHIBA Corporation, 1998
8. V.G. Oklobdzjja, V.M. Stojanovic, D.M. Markovic, N.M. Nedovic, *Digital system clocking high-performance and low-power aspects* (Wiley Interscience, US, 2003)
9. Patent, US20100109707, http://www.freepatentsonline.com/y2010/0109707.html, Accessed on 26 Feb 2011
10. F. Rivoallon, Reducing switching power with intelligent clock gating, Xillix WP370 (V 1.2), 5 Oct 2010
11. P.J. Shoenmakers, J.F.M. Theeuwen, Clock Gating on RT-Level VHDL, in Proceedings of the international Workshop on logic synthesis, Tahoe City, June 7–10, 1998, pp. 387–391
12. Tilera, Tile64 Processor. Tilera Corporation, San Jose, http://www.tilera.com/products/processors/TILE64 (2012), Accessed on Jan 2012
13. V. Tirumalashetty, H. Mahmoodi, *Clock gating and negative edge triggering for energy recovery clock* (ISCAS, New Orleans, 2007), pp. 1141–1144
14. F.J. Wakerly, *Digital design principles and practices* (Prentice Hall, US, 2005)
15. G.K. Yeap, *Practical low-power digital VLSI design* (Kluwer Publishing, UK, 1998)

Software Deployment for Distributed Embedded Real-Time Systems of Automotive Applications

Florian Pölzlbauer, Iain Bate and Eugen Brenner

Abstract Automotive applications can be described as distributed embedded software which perform real-time computation on top of a heterogeneous hardware platform. One key phase in designing distributed software systems is *software deployment*. Therein it is decided how software components are deployed over the hardware platform, and how the communication between software components is performed. These decisions significantly determine the system performance. This chapter tackles the software deployment problem, tailored to the needs of the automotive domain. Thereby, the focus is on two issues: the *configuration of the communication infrastructure* and how to *handle design constraints*. It is shown, how state-of-the-art approaches have to be extended in order to tackle these issues, and how the overall process can be performed efficiently, by utilizing search methodologies.

1 Introduction

In the past, automotive electronics and avionics systems were designed in a federated manner. Most functionality was implemented by special-purpose hardware and hardware-tailored software. One control unit performed only one or at most a limited number of individual functions, and functions had their own dedicated hardware. As the functionality steadily increased, the number of control units has also

F. Pölzlbauer (✉)
Virtual Vehicle, Graz, Austria
e-mail: florian.poelzlbauer@v2c2.at

I. Bate
Department of Computer Science, University of York, York, UK
e-mail: iain.bate@cs.york.ac.uk

E. Brenner
Institute for Technical Informatics, Graz University of Technology, Graz, Austria
e-mail: brenner@tugraz.at

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective*, Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5_12, © Springer-Verlag Berlin Heidelberg 2014

increased. Nowadays cars contain up to 80 control units. During the last several years, a paradigm shift has occurred. The design of electronics has moved from a hardware-oriented to a software/function-oriented approach. This means that functionality is mainly based on software which is executed on general-purpose hardware. In order to enable this trend an interface layer (AUTOSAR [2]) was introduced which separates the application software from the underlying hardware. At the same time, software development steadily moves from hand-coded to model-driven. In model driven development, system synthesis is an important design step to give a partitioning/allocation. The synthesis transforms the Platform Independent Model (PIM) of the system, held in views such as UML's class and sequence diagram, into a Platform Specific Model (PSM), held in views such as UML's deployment diagrams. Design-languages which support model-driven development (such as UML, EAST-ADL, MARTE, etc.) provide dedicated diagrams (e.g.: component, deployment, communication, timing).

In order to deploy the application software onto the execution platform, several configuration steps need to be performed. In the literature this is often referred to as the Task Allocation Problem (TAP). TAP is one of the classically studied problems in systems and software development. It basically involves two parts. Firstly allocating tasks and messages to the resources, i.e. the processors and networks respectively. Secondly assigning attributes to the tasks and messages. Tasks represent software component, and are described by their timing and resource demand (e.g. memory). Messages represent communication between tasks, and are described by their data size and timing. Processors represent the computational units which execute tasks, and are described by their processing power and memory. Networks enable cross-processor communication, and are described by their bandwidth and protocol-specific attributes. In its simplistic form it is an example of the Bin Packing Problem (BPP) where the challenge is to avoid any resource becoming overloading [11]. This "standard" version of the problem is recognised as being NP-hard. Solutions normally involve three components: a means of describing the problem, a fitness function that indicates how close a solution is to solving the problem, and a search algorithm. A wide range of techniques have been used for searching for solutions with heuristic search algorithms, branch and bound, and Integer Linear Programming being the main ones. The problem was later expanded to cover:

1. hard real-time systems where schedulability analysis is used to ensure that the system's timing requirements are met as failing to meet them could lead to a catastrophic effect [3]
2. reducing the energy used [1]
3. making them extensible [19], i.e. so that task's execution times can be increased while maintaining schedulability
4. handling change by reducing the likelihood of change and the size of the changes when this is no longer possible [8]
5. supporting mode changes with the number of transitions minimized [6] and fault tolerance [7].

Open Issues of State-of-the-Art

The list above represents an impressive subset of the problems that need to be solved, however it still does not meet all needs of modern hard real-time systems. There are (at least) two important problems not covered.

- Firstly there are often constraints over the solution, e.g. replicas of tasks cannot reside on the same processor, or that tasks should reside on the same processor near a certain physical device.
- Secondly, communication demand of applications is steadily increasing. Thus, due to limited bandwidth, bus systems are becoming a bottleneck. State-of-the-art bus configuration approaches do not tackle this problem, since they use the bandwidth in an inefficient way. This is due to the fact that frames are hardly utilized, and thus too much overhead data is generated. This not only leads to poor bandwidth usage, but may also lead to unschedulable bus systems. The Frame Packing Problem (FPP) deals with this issue, by packing several messages into a single bus frame, thus improving bandwidth usage and reducing the likelihood of an unschedulable solution when in fact a schedulable one is feasible. In the coming years the communications bus is likely to become a greater bottleneck.

Both these problems have been studied to a limited extent, however this chapter shows how they can be solved more effectively and efficiently (including the scalability to larger more complex systems) using automotive systems as an example.

Outline

The structure of this chapter is as follows. The software deployment problem is outlined in Sect. 2 starting with an explanation of the standard TAP before explaining the needs of hard real-time systems using the domain of automotive systems as an example. Next, a solution to the FPP is presented and demonstrated compared to the previous state-of-the-art approaches. Then, the standard TAP (including the FPP) is extended to deal with the constraints from the automotive industry, before finally directions for future work are outlined.

2 The Software Deployment Problem

The purpose of this section is to explain the **standard TAP**, and outline how it might be solved. The standard TAP refers to the allocation of tasks and messages to processors and networks respectively. Originally it did not consider constraints over and dependencies between tasks.

2.1 The Standard Problem

Assuming a given software application (consisting of several communicating tasks) and a given execution platform (consisting of several processors, connected via

networks). Software deployment (or task allocation) deals with the question, how the software application should be allocated onto the execution platform. Thereby, objectives need to be optimized and constraints need to be satisfied. Applied to hard real-time systems, the process consists of the following design decisions:

- task allocation: local allocation of tasks onto processors
- message routing: routing data from source processor to destination processor via bus-systems and gateway-nodes
- frame packing: packing of application messages into bus frames
- scheduling: temporal planning of the system execution (task execution on processors, frame transmission on bus-systems).

These steps are followed by system performance and timing analysis in order to guarantee real-time behaviour. Due to the different design decisions involved, the terms *software deployment* or *task allocation* seem inappropriate to describe the overall process. The term **system configuration** seems more adequate. This is why it will be used throughout this chapter, from this point on.

2.2 Solving the Standard Problem

For several years, system configurations have be designed manually by engineers. To a large degree, this approach is still performed today. Due to the steadily increasing system complexity, the manual approach reaches several limitations:

- It is hard for engineers to keep in mind all direct and indirect interactions (e.g. direct precedence relationships between tasks or indirect relationships such as preemption and blocking) within the system. Design mistakes may occur.
- Generating a system configuration is time consuming. Due to time constraints, only a small number of system configurations are generated and evaluated. Thus it is unlikely that the generated system configuration is close to optimal.

In order to overcome these issues, **design automation** can be applied. Therein, design decisions are performed by algorithms. Thanks to high computation power available, a very high number of system configurations can be generated and evaluated within a reasonable time frame (e.g. several hundred thousand configuration can be evaluated within several hours). This automated approach significantly increases the coverage of the design space, and thus increases the probability of finding near-optimal solutions. Consequently it increases the confidence into the solution. However, due to the enormous design space, complete coverage is impossible. Experience has shown that approaches have to be efficient, in order to be applicable to the typical size and complexity of real world systems [18, 19].

In order to perform system configuration Design Space Exploration (DSE), the meta-heuristic search algorithm *Simulated Annealing (SA)* can be utilized. It is a well known algorithm in the domain of artificial intelligence. Its name and inspiration come from annealing in metallurgy, a technique involving heating and controlled

Algorithm 1: System Configuration Optimization algorithm (based on Simulated Annealing search algorithm)

```
   Input: config.init /* initial configuration */
   Data: t /* temperature */
   Data: iter.max /* max.iterations */
   Data: iter.at.t.max /* max.iterations at constant t */
 1 begin SystemConfigurationSimulatedAnnealing
 2     /* initialize */;
 3     cost.init = cost(config.init);
 4     config.current = config.init /* start at initial configuration */;
 5     cost.current = cost.init;
 6     config.best = config.init;
 7     cost.best = cost.init;
 8     /* search, until stopping-criteria is reached */;
 9     while stop() = false do
10         while iter.at.t < iter.at.t.max do
11             /* propose new configuration */;
12             config.new = neighbour(config.current);
13             cost.new = cost(config.new);
14             /* accept move? */;
15             if acceptMove() = true then
16                 /* improvement of best configuration? */;
17                 if cost.new < cost.best then
18                     /* remember best */;
19                     config.best = config.new;
20                     cost.best = cost.new;
21                 end
22             end
23             iter.at.t++;
24             /* next iteration at constant t */;
25         end
26         cool(t);
27         iter.at.t = 0;
28         /* resume search at lower t */;
29     end
30 end
   Output: config.best /* best configuration found */
```

cooling of a material. It has proven to be very robust, and can be tailored to specific problems with low effort. However, the main reason for using SA is that it is shown in [5] how SA can be tailored to address *system configuration upgrade* scenarios. This aspect will be re-visited in Sect. 7.

In order to apply SA to a specific problem (here: system configuration), the following methods have to be implemented:

- neighbour: Which modification can be applied to a system configuration, in order to get a new system configuration? These represent the modification an engineer would perform manually.

- energy (cost): How "good" is a system configuration? This represents the metrics that are used to evaluate a system configuration.

Algorithm 1 shows the overall procedure. The search starts from an initial configuration, which is either randomly generated or given by the user. By applying the neighbour function, a new configuration is generated. By analysing its cost, the SA determines whether or not to accept it. After a certain number of iterations, the value of parameter t (which represents the temperature in the annealing process) is decreased, which impacts the acceptance rate of *worse* configurations. The overall search stops, due to a defined exit criteria (usually a defined number of iterations or a defined cost-limit).

The following optimization objectives are encoded into the cost function.

- number of needed processors $\rightarrow$ min
- bus utilization $\rightarrow$ min
- processor CPU utilization $\rightarrow$ max and balanced.

The individual terms are combined into a single value, using a scaled weighted sum. Determining adequate weights is challenging, and should be done systematically [15].

$$cost = \frac{\sum w_i \cdot cost_i}{\sum w_i} \tag{1}$$

In order to get a modified system configuration, different neighbour moves can be performed. Most commonly used are:

- re-allocate a task to another processor
- swap the allocation of two tasks (which reside on different processors)
- change scheduling attributes: For priority-based scheduling, changing the priority of a task. For time-triggered scheduling, change the time-slot assignment.

To ensure that the temporal attributes of the system meet the requirements, two analyses need to be performed: Before starting the DSE, worst-case execution time (WCET) analysis must be performed for each task. During the DSE, schedulability analysis must be performed. Therefore, worst-case response time (WCRT) analysis can be applied.

$$r_i = J_i + B_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i + J_j}{T_j} \right\rceil C_j \tag{2}$$

3 The Needs of the Automotive Industry

Electronics and embedded software was introduced into automotive systems in 1958. The first application was the electronically controlled ignition system for combustion engines. It was introduced in order to meet exhaust emission limits. Over the years, a wide range of functionality of a car was implemented by electronics and embedded software. Thereby, the embedded real-time software was developed in a federated manner. Software was tailored to special-purpose hardware, and almost each software-component was executed on a dedicated hardware-unit. As a consequence, the software-components were quite hardware-dependent, which made it hard to migrate software-components to new hardware-units. Also, the number of hardware-units dramatically increased, which increased system cost and system weight.

To overcome these issues, a paradigm shift has occurred during the last several years. Software is executed on general-purpose hardware. Processing power of this hardware steadily increases, thus allowing to execute several software-components on a single hardware-unit. In order to make it easier to migrate software-components to new hardware-units, software-components are separated from the underlying hardware. This is enabled by the introduction of an interface layer, which abstracts hardware-specific details, and provides standardized interfaces to the application-layer software-components. In the automotive domain, the AUTOSAR standard [2] has positioned itself as the leading interface-standard. The main part of AUTOSAR with respect to TAP is the Virtual Function Bus (VFB). This allows two software-components to communicate with each other without explicit naming and location awareness. Basically the source would send a system-signal to the destination via a VFB call. The actual details of where and how to send the signal is decided by the Runtime Environment (RTE), which instantiates the VFB calls on each processor. Taking this approach restricts changes to the RTE when a new task allocation is generated. In other domains, similar trends can be observed. In the avionics domain, the Integrated Modular Avionics (IMA) standard implements a very similar concept referred to as Virtual Channels (instead of VFB calls) and System Blueprints (containing the lookup tables).

Challenges

In order to solve the automotive system configuration problem, the general system configuration approaches need to be tailored to automotive-specific demands. Thereby, the following issues are the most challenging ones. To a large degree, these issues are not sufficiently covered in the literature.

- The configuration of the communication infrastructure needs to tackle details of automotive bus protocols. Thereby, a special focus needs to be set on frame packing.

- Automotive system configuration is subject to a set of heterogeneous design constraints. Only a subset of them is covered in the literature. Efficient methods for ensuring constraint satisfaction are needed.
- Automotive systems are rarely designed "from scratch". Instead, existing systems (or parts of thereof) are taken as an initial solution. These are extended in order to meet current requirements. In addition, system components may be used within several system variants. Consequently, several legacy design decisions may be in place, which must be taken into account during system configuration. This imposes additional constraints.
- The search algorithms need to scale to realistic sizes and complexities of systems. A significant influence on scalability is how hard it is to synthesise a schedulable solution. Broadly speaking the closer the resources are to their utilisation limits the more difficult the challenge.

3.1 Frame Packing

A large number of automotive systems perform control-related tasks. Physical data is sensed by sensors, read by IOs and processed by application software-components. The output data is then fed back into the physical world via IOs and actuators. Due to cost constraints, the accuracy of sensors, IOs and actuators is limited to 8–12 bits. As a consequence, most data is encoded by 1 to 16 bits variables.

Most automotive bus-systems transmit bus frames which can contain up to 8 byte payload data (LIN, CAN). Emerging bus-systems like FlexRay and automotive Ethernet can transmit more payload data (e.g. 254 byte for FlexRay). However, in addition to the payload data, protocol-related data (header, checksum,…) must be transmitted. For LIN and CAN these protocol data consume 64 bits per frame. Based on the packing density of a frame, the bandwidth efficiency is between 1, 5 % $(1 : 1 + 64)$ and 50 % $(64 : 64 + 64)$. Over the years, the communication demand of automotive systems has steadily increased. However, the available bus bandwidth is a limited resource. Due to physical constraints, the bandwidth of a shared bus cannot easily be increased. As a consequence, bandwidth-efficient frame packing is needed. This is why frame packing is heavily used in the automotive domain by engineers. Typically, frames contain between 48 and 64 bits payload. However, almost no work in the literature on system configuration considers this issue. Instead, almost all works consider only the following options:

- If a message fits into a frame (message.size $\leq$ frame.payload.max), each message is packed into a separate frame.
- If a message does not fit into a frame (message.size $>$ frame.payload.max), the message is split into several parts, each part is packed into a separate frame, and the parts are re-joined at the receiving processor.

These simplistic assumptions lead to several negative attributes of the communication configuration: First, it is a non-realistic approach, which is not accepted by

industry. Second, the approach leads to a high number of frames. The frame-set has poor bandwidth usage, since too much bandwidth is consumed by the protocol-overhead data. The high number of frames also leads to increased interference between frames, thus leading to increased response times (and decreased schedulability). In order to overcome these issues, realistic frame packing must be performed by system configuration approaches. Therefore, several messages must be packed into a single bus frame. Packing objective should be to minimise the bandwidth demand of the resulting frame-set.

The Frame Packing Problem (FPP) is defined as follows: A set of messages $M = \{m_1, m_2, \ldots, m_n\}$ must be packed into a set of bus frames $F = \{f_1, f_2, \ldots, f_k\}$, subject to the constraint that the set of messages in any frame fits that frame's maximum payload. Usually, the FPP is stated as an optimization problem. The most common optimization objectives are: (1) minimize the number of needed frames; or (2) maximize the schedulability of the resulting frame-set. A message is defined by $m_i = [s_i, T_i, D_i]$. A frame is define by $f_j = [pm_j, M_j, T_j, D_j]$. In general each frame may have its individual max.payload (depending on the bus protocol). However, usually all frames on the same bus have the same max.payload. Symbols are explained in Table 2.

3.2 Design Constraints

Design constraints may have a wide variety of sources. Most relevant are:

- safety considerations: If safety analysis of the entire system has been performed (e.g. hazard and risk analysis, in accordance with ISO 26262 [10]), safety requirements can be derived. These impose constraints on design decisions.
- compatibility to legacy systems: Automotive systems are usually designed in an evolutionary fashion. A previous version of the system is taken as a starting point and is extended with additional features, in order to satisfy current demands/requirements. Thus, legacy components may impose constraints on design decisions.
- engineer's experience: Engineers who have been designing similar systems typically have figured out "best practices". These may exclude certain design decisions, thus imposing additional constraints.
- legal requirements: Certain design solutions may not be allowed, in order to comply to legal regulations.

Within the AUTOSAR standard, design constraints that might occur have been specified in the *AUTOSAR System Template*. Therein, a variety of constraint-types can be found. However, these constraints are not only relevant for automotive systems, and could easily be applied to other domains (e.g. rail, aerospace, automation,…). Table 1 provides a summary of the constraint-types. They can be categorized within 6 classes.

Table 1 Constraint-types specified within AUTOSAR system template

Constraint-class	Constraint-type	Literature
A: limited resources	A-1: processor CPU speed	Yes
	A-2: processor memory	Yes
	A-3: bus bandwidth	Yes
B: real-time behaviour	B-1: task deadline	Yes
	B-2: communication deadline	Yes
	B-3: end-to-end deadline	Yes
C: allocation (task to processor)	C-1: dedicated processors	Yes
	C-2: excluded processors	Yes
	C-3: fixed allocation	Yes
D: dependencies (task to task)	D-1: grouping	No*
	D-2: separation	Yes
E: data routing (data to bus)	E-1: processor-internal only	No*
	E-2: dedicated buses	No
	E-3: excluded buses	No
	E-4: same bus	No
	E-5: separated buses	No
F: frame packing (data to frame)	F-1: dedicated frame	No
	F-2: same frame	No
	F-3: separated frames	No

*Not stated as a constraint, but used as means to reduce bus utilization

Since all embedded software must content itself with limited resources, these constraints are well studied in the literature. Automotive systems must be reliable, and thus have to satisfy additional constraints. Most safety-related functions must guarantee real-time behaviour, especially if human life is at risk (e.g. drive-by-wire application in a car). If the function is high-critical, it may be needed to apply redundancy. Therefore replicated tasks must not reside on the same processor (task separation), certain processors are inadequate for handling certain tasks (excluded processors), and data must be transferred via separated buses, probably even within separated bus frames.

It is interesting to note, that several constraint-types are not addressed in the literature. Especially constraints that focus on the configuration of the communication infrastructure have not been tackled. This can be explained, because most works on system configuration (e.g. task allocation) use simplified models for cross-processor communication. These models do not cover all relevant details of the communication infrastructure, and thus the use of detailed constraints seems obsolete. In automotive systems though, these constraints are of high importance.

4 Solving the Frame Packing Problem

The FPP can be seen as a special case of the Bin Packing Problem (BPP), which is known to be a NP-hard optimization problem. In the literature there are several heuristics for the BPP [4]. Well known on-line heuristics are: next fit, first fit, best fit, etc. Off-line heuristics extend these approaches by applying initial sorting, resulting in: next fit decreasing, best fit decreasing, etc. In general, off-line approaches outperform on-line approaches, since off-line approaches can exploit global knowledge, whereas on-line approaches have to take decisions step-by-step, and decisions cannot be undone.

Inspired by the main concepts of BPP heuristics, heuristics for the FPP have been developed. It is interesting to note that there are only a few works in the literature addressing the FPP, although the FPP has significant impact on the performance of the system. Most FPP algorithms mimic some BPP heuristic. Sandström et al. [17] mimics *next fit decreasing*, where messages are sorted by their deadline. Saket and Navet [16] mimics *best fit decreasing*, where messages are sorted by their periods. In addition, the sorted message-list is processed alternately from the beginning and the end. In [14] messages are sorted by their offsets. References [14, 16] combine the FPP with the scheduling problem. References [18, 19] include the FPP into the TAP. Thereby FPP and TAP are formulated as a Mixed Integer Linear Problem (MILP), and solved sequentially

Table 2 Symbols used for frame packing

Symbol	Description
m	Message
f	Frame
M	Set of messages
F	Set of frames
D	Deadline
T	Period
T_m	Period of message
T_f	Period of frame
s_m	Data size of message
pay_f	Payload of frame
pm	Max. payload of frame
oh_f	Overhead of frame
br	Baudrate of bus
bw	Bandwidth demand

4.1 Insufficiencies of State-of-the-Art Approaches

Besides these differences, all state-of-the-art FPP algorithms share one common issue: The packing decision is made based on one condition only:

$$\text{message.size} \leq \text{frame.payload.left} \tag{3}$$

Due to limited bus bandwidth, frame packing should be bandwidth demand minimizing. The bandwidth demand of a frame is determined by two factors: data (payload and overhead) and period.

$$bw_f = \frac{pay_f + oh_f}{T_f} \tag{4}$$

The payload contains all packed-in messages. The overhead contains all protocol-specific data. Since a frame may have several messages packed-in, the frame must be transmitted at a rate which satisfies the rate of all packed-in messages. Thus the lowest message period determines the frame period.

$$T_f = \min_{i\,=\,m \in f} \{T_i\} \tag{5}$$

In order to achieve minimal bandwidth demand, the following aspects must be tackled:

1. The number of frames must be minimized, in order to minimize the overhead data.
2. Messages which are packed into the same frame should have *similar* periods, in order to avoid sending a message more frequently than needed.

Some state-of-the-art FPP approaches try to tackle these.

1. Messages are packed into frames, for as long as there is space left. This reduces the number of needed frames, and thus the bandwidth consumption by the overhead data.
2. Messages may be sorted by their period, before performing the packing. This way, the period variation is reduced. Thus, messages with *similar* periods are packed into the same frame.

Although both strategies (dense packing and initial sorting) may help reducing bandwidth demand, they cannot guarantee minimal bandwidth demand. Here is the **flaw**: During the packing procedure, only the necessary condition (see Eq. 3) is considered. Instead, each packing step must be subject to optimality considerations.

4.2 Optimality Criteria for Frame Packing

Sophisticated frame packing should be bandwidth demand minimizing. The optimal/ideal situation is to have fully utilized frames and all messages inside a frame having the same (or very similar) periods. In general (since both the data size as well as the period of messages varies) the real situation represents a trade-off: If messages with different periods are packed into a frame, the frame must be sent at the lowest period, and thus some of the messages are sent more frequently than needed. This increases the bandwidth consumption. On the other hand, the more messages are packed into a frame, the less frames are needed. Thus less overhead data is sent. This reduces the bandwidth consumption. The optimal trade-off can be accomplished as follows:

Assume the following minimal example: There exists a frame that already has some messages packed-in. Another message needs to be packed-in and it can fit the existing frame. The question is: Should the message be packed into the existing frame (thus extending it), or should the message be packed into a new frame? This decision can be taken in an optimized way, by analysing the bandwidth demand of the two alternatives (left and right side of equation):

$$\underbrace{\frac{pay_f + s_m + oh_f}{T'_f}}_{\text{extended frame}} = \underbrace{\frac{pay_f + oh_f}{T_f}}_{\text{existing frame}} + \underbrace{\frac{s_m + oh_f}{T_m}}_{\text{new frame}} \tag{6}$$

Note that the period of a frame is determined by the message with the lowest period inside the frame. By adding a message, the period of the extended frame T'_f may change. Originally it is T_f.

$$T'_f = \min\left\{T_f \cup T_m\right\} \tag{7}$$

Depending on the relation between T_m and T_f, there are 3 cases for this packing situation. For each of them, an optimal decision can be made.

Case I: $T_m = T_f$

If the periods of the frame and the message are equal, it is always beneficial to extend an existing frame. Creating a new frame is never beneficial, because of the additional overhead data.

$$\frac{pay_f + s_m + oh_f}{T} = \frac{pay_f + oh_f}{T} + \frac{s_m + oh_f}{T} \tag{8}$$

$$oh_f < 2\ oh_f \tag{9}$$

Case II: $T_m > T_f \Rightarrow T'_f = T_f$

The trade-off is: By extending the frame, the message will be sent more frequent than needed, but no additional overhead is created. By creating a new frame, additional overhead is created, but the message will not be sent too frequent.

$$\frac{pay_f + s_m + oh_f}{T_f} = \frac{pay_f + oh_f}{T_f} + \frac{s_m + oh_f}{T_m} \tag{10}$$

$$\frac{s_m}{T_f} = \frac{s_m + oh_f}{T_m} \tag{11}$$

At the threshold period of the message, the two alternatives perform equally.

$$T^*_m = T_f \frac{s_m + oh_f}{s_m} \tag{12}$$

Thus, the optimal solution is:

- $T_m < T^*_m \Rightarrow$ extending the frame is beneficial
- $T_m > T^*_m \Rightarrow$ creating a new frame is beneficial

Case III: $T_m < T_f \Rightarrow T'_f = T_m$

The trade-off is: By extending the frame, the frame will need to be sent more frequent, but no additional overhead is created. By creating a new frame, the original frame will not be sent more frequent, but additional overhead is created.

$$\frac{pay_f + s_m + oh_f}{T_m} = \frac{pay_f + oh_f}{T_f} + \frac{s_m + oh_f}{T_m} \tag{13}$$

$$\frac{pay_f}{T_m} = \frac{pay_f + oh_f}{T_f} \tag{14}$$

The threshold period is:

$$T^*_m = T_f \frac{pay_f}{pay_f + oh_f} \tag{15}$$

Thus, the optimal solution is:

- $T_m < T^*_m \Rightarrow$ creating a new frame is beneficial
- $T_m > T^*_m \Rightarrow$ extending the frame is beneficial.

4.3 Improved Frame Packing Heuristic

The main issue of state-of-the-art frame packing heuristics is: During packing only the necessary packing condition (see Eq. 3) is checked. In case the periods of messages vary significantly, the approaches perform poorly, even if messages are sorted by their periods.

To overcome this issue, the packing decision must be taken by also incorporating the trade-off optimality criteria, derived above. The proposed frame packing heuristic (see Algorithm 2) incorporates these criteria. Its structure is inspired by the *Fixed Frame Size* approach of [17] which mimics *next fit decreasing*. However, messages are not sorted by their deadline. Instead messages are sorted by their period, inspired by Saket and Navet [16]. However, the packing procedure is not done in a bi-directional way.

Algorithm 2: Frame packing (based on optimal decisions)

```
   Input: messages
 1 sort(messages, T, increasing) /* sort by T [0..n] */;
 2 frame = new frame;
 3 foreach message do
 4 |  if frame.payload.left ≥ message.size then
 5 |  |  /* take most beneficial decision */;
 6 |  |  benefit = extendOrNew(message, frame);
 7 |  |  if benefit = extend then
 8 |  |  |  pack(message, frame);
 9 |  |  else if benefit = new then
10 |  |  |  pack(message, new frame);
11 |  |  end
12 |  else
13 |  |  pack(message, new frame);
14 |  end
15 end
   Output: frames
```

Within the *ExtendOrNew* method, the most beneficial decision is determined using the optimality criteria presented earlier. This way each packing step has minimal increase of bandwidth demand.

Due to the NP-hard nature of the FPP, the proposed approach cannot guarantee an optimal packing. However, experimental evaluation shows that it outperforms state-of-the-art approaches. On the one hand, the bandwidth demand of the resulting frame-set is significantly decreased. On the other hand, the schedulability of the frame-set is less sensitive against timing uncertainties.

Table 3 shows the improvements in bandwidth demand. On the left side, improvements are shown due to *number of sending processors* and *bus baudrate*. The main improvements can be seen for systems with higher number of sending processors. Such systems will be used in future automotive applications. On the

Table 3 Improvement of Poelzlbauer et al. compared to state-of-the-art

# nodes	Improvement (%)			Message (bit)	Improvement (%)
	125 k	256 k	500 k		
1…3	0.0	0.0	0.0	1…8	0…18
5	5.9	2.3	0.0	1…16	0…18
10	14.4	6.2	3.3	1…24	0…19
15	13.8	15.0	2.4	1…32	0…16
20	17.6	16.2	6.2	1…64	0…6

right side, improvements are shown due to *message size*. An interesting finding is that the improvements are almost the same for a wide range of message sizes. Currently, physical data is mainly encoded in up to 16 bit variables. Future applications may need higher accuracy, thus 32 bit variables may be used. The proposed approach also handles these systems in an efficient way. More details on the evaluation can be found in [13].

5 Handling Constraints

The task of finding a system configuration is challenging and time-consuming. By applying design automation, and thus using search algorithms, a large number of potential configurations can be evaluated within reasonable time. However, finding a system configuration for industrial applications is subject to a set of heterogeneous constraints. Table 1 gives an overview. Thus the question arises: How can these constraints be handled and satisfied?

In order to determine, how many of the constraints are satisfied (and how many are violated) by a configuration, the *cost function* of the SA search framework needs to be extended. Therefore, the cost term *constraint violations* is added. It is stated as a minimization term. Configurations which violate constraints are punished. Ideally no constraints are violated. Due to the heterogeneous nature of the constraints, different constraint-types may be of different importance. This can be addressed by applying a weighted sums approach. Each constraint-type is evaluated as:

$$cost_i = \frac{\#\text{ of elements that violate a constraint-type}}{\#\text{ of elements that have a constraint-type associated}} \tag{16}$$

Basically, this extended SA search framework should be able to find system configurations which satisfy all constraints. Since constraint violations are punished, the search should be directed towards regions of the design space where all constraints are satisfied. However, experimental evaluation reveals some more diverse findings. For some constraint-types, this approach works. Configurations are modified, until the number of constraint violations becomes quite low. The approach works quite well for *E-1 processor-internal only*. This can be explained by the fact, that this

constraint-type is in alignment with another optimization target (bus utilization → min). For some other constraint-types (e.g. *C-1 dedicated processor*) the approach is able to reduce the number of constraint violations, but cannot eliminate them. In addition, it takes a lot of search iterations, until the number of constraint violations drops. For other constraint-types (e.g. *F-2 same frame*) the approach fails entirely.

Concluding, this approach is inefficient and ineffective, and thus is not applicable to industrial system configuration instances. The question arises: How can the set of heterogeneous constraints be handled and satisfied in an efficient way?

5.1 Improving Efficiency

In order to overcome these issues, and to handle constraints in an efficient way, two issues must be addresses:

1. neighbour: The neighbour-moves are not aware of the constraints. Thus infeasible configuration may be generated. However, neighbour-moves should be aware of the constraints, and only generate configuration within feasible boundaries.
2. pre-conditions: In order to be able to satisfy a constraint, a set of pre-conditions may need to be fulfilled (e.g. certain packing constraints need certain routing conditions). Thus it is highly important to fulfill these pre-conditions, else constraints can never be satisfied.

It is advised to split the entire system configuration problem into several sub-problems, and to handle design decisions and constraints within these sub-problems. Considering the various interactions, the following sub-problems seem appropriate:

- task allocation and message routing
- frame packing
- scheduling.

5.1.1 Task Allocation and Message Routing

In order to increase the efficiency of the search, the neighbour-moves for task allocation are modified as follows: Each task has a set of *admissible processors* associated. Only processors out of these sets are candidates for allocation modifications.

The question is: How can the sets of admissible processors be derived, so that all allocation- and routing-constraints are satisfied? To achieve this, a set of rules are derived and applied. Most of these rules are applied before the search-run. Thus it is a one time effort.

E-1: By grouping the sender- and the receiver-task (forming a task-cluster), it can be ensured that the task allocation algorithm will allocate both tasks to the same processor. Thus, the communication between these tasks is always performed processor-internal.

E-2 and E-3: Based on these sets, a set of admissible buses can be calculated for each message.

$$B_{adm} = \begin{cases} B \setminus B_{ex} & \text{if} \quad B_{ded} = \{\} \\ B_{ded} \setminus B_{ex} & \text{otherwise} \end{cases} \tag{17}$$

This admissible message-routing implies a set of admissible processors X for the sender- and receiver-task of this message. Only processors connected to the admissible buses of the message are potential candidates for hosting the sender- and receiver-task.

$$P_{adm}^{(t \to m \to t)} = P \text{ connected to } B_{adm} \tag{18}$$

Since a task may send and receive several messages, only the intersected set X is a potentially admissible processor for each task.

$$X = \bigcap P_{adm}^{(t \to m \to t)} \tag{19}$$

E-4: Two messages can only be routed via the same bus, if their sender-tasks reside on the same processor and also their receiver-tasks reside on the same processor. Thus, E-4 can be satisfied by two D-1 constraints.

C-1 and C-2: Based on these sets, a set of admissible processors can be calculated for each task. Thereby, the set of admissible buses (derived from E-2 and E-3) of the sent/received messages has also to be taken into account.

$$P_{adm} = \begin{cases} (P \cap X) \setminus P_{ex} & \text{if} \quad P_{ded} = \{\} \\ (P_{ded} \cap X) \setminus P_{ex} & \text{otherwise} \end{cases} \tag{20}$$

D-1: Similar to E-1, this constraint can be resolved by grouping the associated tasks (forming a task-cluster). If tasks are grouped, the set of admissible processors for a task cluster c is:

$$P_{adm}^{(c)} = \bigcap_{t \in c} P_{adm} \tag{21}$$

D-2: The set of admissible processors can be updated dynamically (during the design space exploration).

$$P_{adm.dyn} = P_{adm} \setminus P_{ex.dyn} \tag{22}$$

$$P_{ex.dyn} = P \text{ of tasks that the current task must be separated from} \tag{23}$$

C-3: If an allocation is fixed, the task allocation algorithm will not modify that allocation.

5.1.2 Frame Packing

In order to satisfy frame packing constraints, both aspects need to be tackled. On the one hand, the frame packing heuristic needs to be constraint-aware. On the other hand, the necessary pre-conditions (task allocation and message routing) have to be fulfilled. Within this context, pre-condition fulfillment is crucial:

F-2: Two messages can only be packed into the same frame, if both messages are sent from the same processor and routed via the same bus. This can be stated by E-4.
F-1: Similar to F-2, a message can only be packed into the dedicated frame, if they are sent from the same processor and routed via the same bus.

Assuming the pre-conditions are fulfilled, the frame packing constraints can be satisfied by the following constraint-aware packing heuristic. It consists of 4 phases, which are performed sequentially:

Phase 1: F-1: Dedicated frame packing is typically used, because the same *frame catalogue* is used within different cars. To satisfy this constraint, messages that have this constraint associated, will only be packed into the dedicated frame.
Phase 2: F-3: Messages which have this constraint associated are packed into separated frames each.
Phase 3: F-2: Messages which have this constraint associated are packed into the same frame.
Phase 4: All remaining messages (which have no constraint associated) can be packed according to the packing algorithm presented in Sect. 4.

5.2 Implications on Design Space Exploration

Based on the considerations and rules presented earlier, design space exploration can be performed more efficiently. Exploration steps (performed via neighbour moves) are performed based on the following principles:

- Task clusters are treated as single elements during task allocation. Therefore, if a task cluster is re-allocated, all tasks inside that task cluster will be re-allocated to the same processor.
- When picking a "new" processor for a task/task cluster, only processors from the set of admissible processors are used as candidates.
- Frame packing is performed due to the constraint-aware packing heuristic.

As a consequence, a large number of infeasible configurations is avoided, since constraints are not violated. Thus, the efficiency of the search increases. In addition, constraint satisfaction can be guaranteed for certain constraint-types.

Unfortunately, not all constraint-types can be resolved. For a set of constraint-types (A-1, A-2, A-3, B-1, B-2, B-3, E-5) no rules how to satisfy them, could be

Table 4 Constraint encoding as "mandatory" or as "desired"

Type	Mandatory	Desired	Rationale
A-1	x		Utilization $\leq$ 100 % required for schedulability
A-2		x	Utilization $\leq$ 100 % not required for schedulability
A-3	x		Utilization $\leq$ 100 % required for schedulability
B-1		x	Guide search through un-schedulable regions
B-2		x	Guide search through un-schedulable regions
B-3		x	Guide search through un-schedulable regions
E-5	xx	x	Depending on source of constraint (e.g. safety analysis)

Note All other constraint-types can be resolved. Thus they are always satisfied, and don't need to be encoded into the search algorithm any more. Options marked as "xx" represent the option preferred by the authors

derived. These constraints are addressed by the cost-term *constraint violations*. Thereby they can either be represented as a *mandatory* or as *desired*. The following implications should be taken into account, when deciding between these options:

- mandatory: If a mandatory constraint is violated, the configuration is treated as being *infeasible*. Thus it will be rejected. Consequently, the configuration is not considered as the starting point for generating new configurations.
- desired: A configurations that does not satisfy a desired constraint is not rejected. Instead it is punished by a high *cost value*. However, the configuration can still be picked as the starting point for subsequent exploration steps.

The difference may sound minor, but actually has significant impact on the DSE. Using *desired constraints* enables the search to gradually traverse through infeasible regions. However, even configurations with "moderate" cost may be infeasible. Using *mandatory constraints* ensures that all constraints are satisfied for feasible configurations.

Table 4 provides a proposal, in which way each constraint-type could be encoded. The proposal tries to tackle the nature of the constraint-types as well as efficiency considerations, in order to find the most appropriate encoding for each constraint-type.

By resolving constraints and using constraint-aware neighbour moves, the design space exploration can be performed more efficiently. Experimental evaluation shows that the improvements are two-fold: On the one hand, fewer constraints are violated during the search. On the other hand, the *best obtained solution* has improved attributes due to the optimization targets. Figure 1 shows the number of constraint violations during a search-run (for different constraint-types). It evidences the efficiency of the proposed approach of constraint resolving: Without resolving (just using the cost-term to punish a constraint violation) the number of violations is very high. With resolving, the number of violations is significantly lower, or in several cases even zero. More details (especially on the impact on the *best obtained solution*) can be found in [12].

6 Applicability to Real-World Systems

The presented methodologies in this chapter are described tool-independent. However, in order to be applicable for engineers, the methodologies need to be incorporated into state-of-the-art automotive engineering tools. Consequently, the system model (consisting of tasks, messages, processors and networks) needs to be represented in an appropriate format. This could either be a higher-level one (e.g. EAST-ADL) or a more detailed one (e.g. AUTOSAR). The generated outputs of the methodologies are again represented in such formats. Network-specific outputs, such as frame packing, could be encoded into the FIBEX format (which is the state-of-the-art format for automotive networks). Software allocation and scheduling information could be represented in the AUTOSAR format.

The computational model assumes that data produced by a task may trigger the execution of a task which reads the data. If this was not being done, the generating task could produce another instance of the data, and overwrite the older ones. Thus the implementation of the system must ensure that these receiving tasks are executed at appropriate rates and in an appropriate order. Such implementation-specific details need to be addressed in the AUTOSAR RTE generation process (which is not covered in this chapter).

7 Future Research Directions

Automotive electronics and embedded software are developed in accordance to the following life cycle:

- Every few years, a new hardware-platform is developed. Here, new technology may be introduced, especially at the hardware-level. E.g. new bus protocols (like FlexRay or automotive Ethernet) and new micro-controllers (e.g. multi-core

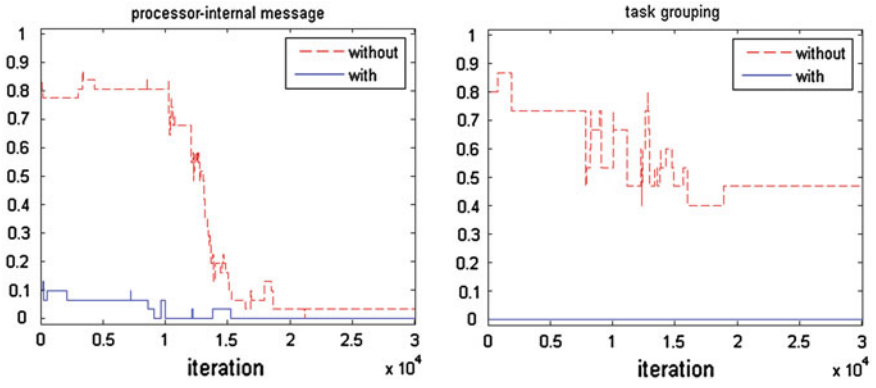


Fig. 1 Constraint satisfaction efficiency without and with constraint resolving

architectures). The goal of this phase is to find a platform which is extensible for future requirements. Within this phase, almost all design decisions may be modified. This phase may be called *finding a system configuration which maximizes extensibility*.

- During the following years, this system configuration is used as the basis. New technologies are rarely introduced. Modifications to the system configuration may mainly be due to 2 scenarios:
  - Minor modifications are applied to the system configuration. The goal is to improve the system. This may be called *system configuration improving*. Thereby, most design decision taken for the initial configuration must be treated as constraints.
  - Additional components (e.g. software) are added to the system, in order to meet new requirements. The goal is to find a system configuration for the extended system. This may be called *system configuration upgrade*. Thereby almost all design decisions from the initial system must be treated as constraints. In addition, the new configuration should be similar to the initial configuration, in order to reduce effort for re-verification.

Consequently, future research activities have to be two-fold: On the one hand, emerging technologies such as multi-core architectures and automotive Ethernet have to be tackled. On the other hand, it must be investigated how these development-scenarios can be addressed. Concerning the latter, basically most ingredients are already at hand.

- In order to find a platform configuration, which is extensible for future modifications/extensions, the key issue is to have test-cases (i.e. software architectures) which represent possible future requirements. This can be addressed by using *change scenarios* [8]. In addition, the configurations can be analysed with respect to *parameter uncertainties*. Well known approaches use sensitivity analysis for task WCET [19]. This can be extended towards other parameters (e.g. periods), thus resulting in multi-dimensions robustness analysis [9].
- The second issue is to actually perform a system configuration modification. A typical improvement scenario could be: reassign priorities to tasks on a certain processor, in order to fix timing issues. Therefore, state-of-the-art optimization approaches could be used, e.g. SA. Of course, the neighbour-moves must be constraint-aware.
- Within a system configuration upgrade, both the software-architecture as well as the hardware-architecture may be subject to changes. Typically new additional software-components (and communication between software-components) are added. In order to provide sufficient execution resources, additional processors may be needed. These scenarios can be addressed as follows: In [6] it is shown, how a system configuration can be found for multi-mode systems. Thereby, the goal is to have minimal changes between modes, thus enabling efficient mode-switches. If the different versions of the system (initial system, extended system) are treated as *modes*, similar methods can be used. However, there is one signifi-

cant difference: Emberson and Bate [6] assumed that the hardware-platform is the same for all modes. In a system configuration upgrade scenario, this assumption is no longer valid.

- Thus, when performing system configuration modifications and system configuration extensions, the key issue is to deal with legacy decisions. These must be treated as constraints. Therefore, constraint handling is needed. This can be addressed by the methodology presented in Sect. 5.

In order to address and solve system configuration upgrade scenarios, the following next steps have to be performed: First, a metric for determining *changes* has to be derived, and tailored to automotive needs. Second, the approach in [6] needs to be extended, so that it can handle changes in the hardware-platform. Third, the *constraint handling* approach needs to be incorporated with the part that will deal with changes.

Acknowledgments The authors would like to acknowledge the financial support of the "COMET K2—Competence Centres for Excellent Technologies Programme" of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), the Austrian Federal Ministry of Economy, Family and Youth (BMWFJ), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG). We also thank our supporting industrial (AVL List) and scientific (Graz University of Technology) project partners.

References

1. T. AlEnawy, H. Aydin, Energy-aware task allocation for rate monotonic scheduling, in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (2005), pp. 213–223
2. AUTOSAR (automotive open system architecture), http://www.autosar.org Revision 4.0. Accessed 9 Sept 2012
3. A. Burns, M. Nicholson, K. Tindell, N. Zhang, Allocating and scheduling hard real-time tasks on a point-to-point distributed system. Workshop on Parallel and Distributed, Real-Time Systems (1993), pp. 11–20
4. E.G. Coffman, M.R. Garey, D.S. Johnson, Approximation algorithms for bin packing: a survey, in *Approximation Algorithms for NP-hard Problems*, Chap. 2 (PWS Publishing Co., Boston, MA, USA, 1996), pp. 46–93
5. P. Emberson, Searching For Flexible Solutions To Task Allocation Problems. PhD thesis, University of York, Department of Computer Science, 2009
6. P. Emberson, I. Bate, Minimising task migration and priority changes in mode transitions, in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (2007) pp. 158–167
7. P. Emberson, I. Bate, Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems, in *IEEE Real-Time Systems Symposium (RTSS)* (2008), pp. 270–279
8. P. Emberson, I. Bate, Stressing search with scenarios for flexible solutions to real-time task allocation problems. IEEE Trans. Softw. Eng. **36**(5), 704–718 (2010)
9. A. Hamann, R. Racu, R. Ernst, Multi-dimensional robustness optimization in heterogeneous distributed embedded systems, in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2007), pp. 269–280
10. ISO 26262, *Road Vehicles—Functional Safety*. Revision 1.0

11. D.S. Johnson, M.R. Garey, A 71/60 theorem for bin packing. J. Complex. **1**(1), 65–106 (1985)
12. F. Pölzlbauer, I. Bate, and E. Brenner. Efficient constraint handling during designing reliable automotive real-time systems. International Conference on Reliable Software Technologies (Ada-Europe) (2012), pp. 207–220
13. F. Pölzlbauer, I. Bate, E. Brenner, Optimized frame packing for embedded systems. IEEE Embed. Syst. Lett. **4**(3), 65–68 (2012)
14. P. Pop, P. Eles, Z. Peng, Schedulability-driven frame packing for multicluster distributed embedded systems. ACM Trans. Embed. Comput. Syst. **4**(1), 112–140 (2005)
15. S. Poulding, P. Emberson, I. Bate, J. Clark, An efficient experimental methodology for configuring search-based design algorithms, in *IEEE High Assurance Systems Engineering Symposium (HASE)* (2007) , pp. 53–62
16. R. Saket, N. Navet, Frame packing algorithms for automotive applications. Embed. Comput. **2**(1), 93–102 (2006)
17. K. Sandström, C. Norström, and M. Ahlmark. Frame packing in real-time communication. International Conference on Real-Time Computing Systems and Applications (RTCSA) (2000), pp. 399–403
18. W. Zheng, Q. Zhu, M. Di Natale, A. Sangiovanni-Vincentelli, Definition of task allocation and priority assignment in hard real-time distributed systems, in *IEEE Real-Time Systems Symposium (RTSS)* (2007), pp. 161–170
19. Q. Zhu, Y. Yang, E. Scholte, M. Di Natale, A. Sangiovanni-Vincentelli, Optimizing extensibility in hard real-time distributed systems, in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2009), pp. 275–284

Editors Biography

Mohammad Ayoub khan is working as Associate Professor, Department of Computer Science and Engineering, School of Engineering and Technology, Sharda University, Plot No. 32-34, Knowledge Park III, Greater Noida, UP 201306, India with interests in Radio Frequency Identification, microcircuit design, and signal processing, NFC, front end VLSI (Electronic Design Automation, Circuit optimization, Timing Analysis), Placement and Routing in Network-on-Chip, Real Time and Embedded Systems. He has more than 11 years of experience in his research area. He has published more than 60 research papers in the reputed journals and international IEEE conferences. He is contributing to the research community by various volunteer activities. He has served as conference chair in various reputed IEEE/Springer international conferences. He is a senior member of professional bodies of IEEE, ACM, ISTE and EURASIP society. He may be reached at ayoub@ieee.org, +91-9871632098.

Saqib Saeed is an assistant professor at the Department of Computer Science Shangrila Road, Sector E-8, Bahria University Islamabad, PIN/ZIP Code: 44000, Pakistan. He has a Ph.D. in Information Systems from University of Siegen, Germany, and a Master's degree in Software Technology from Stuttgart University of Applied Sciences, Germany. He is also a certified software quality engineer from American Society of Quality. His research interests lie in the areas of human-centered computing, computer supported cooperative work, empirical software engineering and ICT4D. He may be reached at saqib.saeed@gmail.com.

Ashraf Darwish received the Ph.D. degree in computer science from Saint Petersburg State University, Russian Federation in 2006 and is currently an assistant professor at the Faculty of Science, Helwan University, Cairo, P.O. 1179, Egypt. Dr. Darwish teaches artificial intelligence, information security, data and web mining, intelligent computing, image processing (in particular image retrieval, medical imaging), modeling and simulation, intelligent environment, body sensor networking.

M. A. Khan et al. (eds.), *Embedded and Real Time System Development: A Software Engineering Perspective,* Studies in Computational Intelligence 520,
DOI: 10.1007/978-3-642-40888-5, © Springer-Verlag Berlin Heidelberg 2014

Dr. Darwish is an editor and member of many computing associations, such as IEEE, ACM, EMS, QAAP, IJCIT, IJSIEC, IJIIP, IJITNA, IJCISIM, SMC, Quality Assurance and Accreditation Authority (Egypt) and a board member of the Russian-Egyptian Association for graduates, and Machine Intelligence Research Lab (MIR Lab) USA.

Dr. Darwish is author of many scientific publications and his publications include papers, abstracts and book chapters by Springer and IGI publishers. He keeps in touch with his mathematical background through his research. His research, teaching and consulting mainly focuses on artificial intelligence, information security, data and web mining, intelligent computing, image processing, modeling and simulation, intelligent environment, body sensor networks, and theoretical foundations of computer science. He may be reached at modarwish@yahoo.com, Tel/Fax: +2 2555 2468.

Ajith Abraham received the Ph.D. degree in Computer Science from Monash University, Melbourne, Australia. He is currently the Director of Machine Intelligence Research Labs (MIR Labs), Scientific Network for Innovation and Research Excellence (SNIRE), P.O. Box 2259, Auburn, Washington, DC 98071, USA, which has members from more than 100 countries. He serves/has served the editorial board of over 50 International journals and has also guest edited 40 special issues on various topics. He has authored/co-authored more than 850 publications, and some of the works have also won best paper awards at international conferences. His research and development experience includes more than 23 years in the industry and academia. He works in a multidisciplinary environment involving machine intelligence, network security, various aspects of networks, e-commerce, Web intelligence, Web services, computational grids, data mining, and their applications to various real-world problems. He has given more than 60 plenary lectures and conference tutorials in these areas. He has an h-index of 50+ with nearly 11K citations as per Google Scholar. Since 2008, Dr. Abraham is the Chair of IEEE Systems Man and Cybernetics Society Technical Committee on Soft Computing and also represented the IEEE Computer Society Distinguished Lecturer Programme during 2011-2013.He is a Senior Member of the IEEE, the IEEE Computer Society, the Institution of Engineering and Technology (U.K.) and the Institution of Engineers Australia (Australia), etc. He is actively involved in the Hybrid Intelligent Systems (HIS); Intelligent Systems Design and Applications (ISDA); Information Assurance and Security (IAS); and Next Generation Web Services Practices (NWeSP) series of international conferences, in addition to other conferences. More information at: http://www.softcomputing.net. ajith.abraham@ieee.org, Personal WWW://http://www.softcomputing.net. Postal address for delivering the books: Professor (Dr.) Ajith Abraham, Aster 13C, Skyline Riverdale Apartments, North Fort Gate, Petah, Tripunithura, Kochi, KL 682301, India.