TUCS

Sergey Ostroumov

Agent-Based Management Systems for Many-Core Platforms

Rigorous Design and Efficient Implementation

Turku Centre *for* Computer Science

TUCS Dissertations
No 196, June 2015

Agent-Based Management Systems for Many-Core Platforms

Rigorous Design and Efficient Implementation

Sergey Ostroumov

To be presented, with the permission of the Faculty of Science and Engineering of the Åbo Akademi University, for public criticism in Auditorium Gamma on June 05, 2015, at 12 noon.

Åbo Akademi University
Faculty of Science and Engineering
Joukahaisenkatu 3-5A, 20520, Turku, Finland

2015

Supervised by

Professor Kaisa Sere Associate Professor Marina Waldén	Faculty of Science and Engineering Åbo Akademi University Turku, Finland
Associate Professor Juha Plosila	Department of Information Technology University of Turku Turku, Finland

Reviewed by

Associate Professor	Stefan Hallerstede
Department	Department of Engineering – Software Engineering
University	Aarhus University
City, Country	Aarhus, Denmark

Professor	Hans Hansson
Department	Embedded Systems
University	Mälardalen University
City, Country	Västerås, Sweden

Opponent

Associate Professor	Stefan Hallerstede
Department	Department of Engineering – Software Engineering
University	Aarhus University
City, Country	Aarhus, Denmark

ISBN 978-952-12-3219-0

ISSN 1239-1883

To my family

Моей семье посвящается

“Do not go where the path may lead, go instead where there is no path and leave a trail.”

– Ralph Waldo Emerson

Abstract

Due to various advantages such as flexibility, scalability and updatability, software intensive systems are increasingly embedded in everyday life. The constantly growing number of functions executed by these systems requires a high level of performance from the underlying platform. The main approach to incrementing performance has been the increase of operating frequency of a chip. However, this has led to the problem of power dissipation, which has shifted the focus of research to parallel and distributed computing.

Parallel many-core platforms can provide the required level of computational power along with low power consumption. On the one hand, this enables parallel execution of highly intensive applications. With their computational power, these platforms are likely to be used in various application domains: from home use electronics (e.g., video processing) to complex critical control systems. On the other hand, the utilization of the resources has to be efficient in terms of performance and power consumption. However, the high level of on-chip integration results in the increase of the probability of various faults and creation of hotspots leading to thermal problems. Additionally, radiation, which is frequent in space but becomes an issue also at the ground level, can cause transient faults. This can eventually induce a faulty execution of applications. Therefore, it is crucial to develop methods that enable efficient as well as resilient execution of applications.

The main objective of the thesis is to propose an approach to design agent-based systems for many-core platforms in a rigorous manner. When designing such a system, we explore and integrate various dynamic reconfiguration mechanisms into agents functionality. The use of these mechanisms enhances resilience of the underlying platform whilst maintaining performance at an acceptable level. The design of the system proceeds according to a formal refinement approach which allows us to ensure correct behaviour of the system with respect to postulated properties.

To enable analysis of the proposed system in terms of area overhead as well as performance, we explore an approach, where the developed rigorous models are transformed into a high-level implementation language. Specifically, we investigate methods for deriving fault-free implementations from these models into, e.g., a hardware description language, namely VHDL.

Sammanfattning

På grund av olika fördelar, så som flexibilitet, skalbarhet och uppdaterbarhet integreras mjukvaruintensiva system i allt större utsträckning i våra vardagsliv. Det stadigt ökande antalet funktioner som utförs av de här systemen ställer höga prestandakrav på den underliggande plattformen. Den huvudsakliga metoden för att öka prestandan har varit att höja klockfrekvensen för ett chipp. Det här har emellertid lätt till problem relaterade till energiförbrukning, vilket har gjort att forskningen har skiftat fokus till parallell och distribuerad beräkning.

Parallella flerkärniga plattformar kan tillhandahålla tillräcklig beräkningskapacitet samtidigt som de har låg energiförbrukning. Det här möjliggör parallell exekvering av mycket beräkningsintensiva tillämpningar. Dessa plattformar kommer på grund av deras höga beräkningskapacitet troligen att användas inom många olika tillämpningsområden, från hemelektronik (t.ex. för videobearbetning) till komplexa säkerhetskritiska kontrollsystem. Utnyttjandet av resurserna på sådana här plattformar måste ändå vara effektivt med avseende på prestanda och energiförbrukning. Det höga antalet funktioner som integreras på ett chipp ökar sannolikheten för diverse fel och uppkomsten av så kallade heta punkter (hotspots), som leder till temperaturproblem. Därutöver kan strålning, som är vanligt förekommande i rymden men också kan vara ett problem på marknivå, orsaka transientfel. Det här kan orsaka att en applikation exekveras på ett felaktigt sätt. Det är därför viktigt att utveckla metoder som möjliggör applikationsexekvering med hög effektivitet och resiliens.

Det huvudsakliga målet med den här avhandlingen är att skapa en metod som möjliggör rigorös design av agentbaserade system för flerkärniga plattformar. I designen av ett sådant här system undersöks och integreras diverse mekanismer för dynamisk omkonfigurering i agenternas funktionalitet. Användningen av dylika mekanismer förbättrar den underliggande plattformens resiliens samtidigt som en acceptabel prestandanivå bibehålls. Designen av systemet följer en metod baserad på formell precisering, som gör det möjligt att garantera att systemet fungerar korrekt med avseende på givna egenskaper.

För att kunna analysera det föreslagna systemet med avseende på så kallat areaöverskott (area overhead) och prestanda undersöks en metod för transformation av rigorösa modeller till ett högnivåspråk för implementering. Mer specifikt undersöks metoder för att härleda felfria implementationer i det hårdvarubeskrivande språket VHDL från de här modellerna.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisors, Professor Kaisa Sere, Associate Professor Marina Waldén and Associate Professor Juha Plosila for their guidance, support and expert advice, for a numerous fruitful scientific discussions we have had as well as for the invaluable comments on the papers and the thesis. It has been an honour and pleasure for me to work with them. Furthermore, I would like to thank Professor Vyacheslav Kharchenko for the guidance and support at the beginning of my PhD studies at National Aerospace University "KhAI".

Second of all, I would like to acknowledge Associate Professor Stefan Hallerstede and Professor Hans Hansson for reviewing the thesis and providing valuable comments that allowed me to improve its quality. I am also thankful to Associate Professor Stefan Hallerstede who has kindly agreed to act as an opponent at the public defence of this thesis.

Moreover, I am very thankful to my co-authors, Doctor Leonidas Tsiopoulos and Doctor Pontus Boström who provided me with diverse viewpoints and made their valuable contribution to my work. I would also like to thank Adjunct Professor Linas Laibinis for the expert advice and fruitful discussions on various topics.

I would like to thank all the members of the Distributed Systems Laboratory, in particular, Associate Professor Luigia Petre, Doctor Mats Neovius, Doctor Maryam Kamali and Petter Sandvik. Additionally, I would like to express my gratitude to all the members of the Embedded System Laboratory, particularly, Professor Johan Lilius, Adjunct Professor Elena Troubitsyna, Adjunct Professor Sébastien Lafond, Doctor Johan Ersfolk, Doctor Anton Tarasyuk, Inna Pereverzeva, Wictor Lund, Simon Holmbacka and Sudeep Kanur. Moreover, I would specifically like to thank Robert Slotte and Fredrik Hällis for helping me with the TilePro platform, which enabled quantitative evaluation of the solutions proposed in the thesis.

I would also like to express my sincere appreciation to the members of the Department of Information Technology at University of Turku, particularly, Adjunct Professor Pasi Liljeberg, Doctor Tomi Westerlund, Doctor Masoud Daneshtalab and Doctor Masoumeh Ebrahimi for their fruitful collaboration.

I am also grateful to Professor Ion Petre for his encouragement and Professor Ivan Porres as well as Associate Professor Patrick Sibelius for the interesting discussions we have had.

Moreover, I am very thankful to the administrative personnel of TUCS – Turku Centre for Computer Science, the Faculty of Science and Engineering and the Academic Office at Åbo Akademi University for their support and help, in particular, Tomi Suovuo, Outi Tuohi, Christel Engblom, Nina Hultholm, Tove Österroos, Susanne Ramstedt, Solveig Vaherkylä and Pia-Maria Kallio. Additionally, I would like to thank the technical personnel for their support: Marat Vagapov, Karl Rönnholm, Joakim Storrank and Niklas Grönblom.

I would like to thank TUCS and the Department of Information Technologies at Åbo Akademi University for the funding provided during my doctoral studies. Specifically, I would like to thank the Doctoral Network in Information Technologies and Mathematics program and the Digihybrid project in the EFFIMA program coordinated by FIMECC. I am also honoured and grateful to Nokia Foundation for supporting my work in the form of research scholarship.

Last but not least, I would like to express my deepest gratitude to my family and friends. Particularly, I would like to thank my best friends Miki and Marta Olszewski for uncountable joyful moments that made me feel like home. I am also grateful to Magnus, Irum and Charmi for bringing many positive moments into my life.

I would specifically like to thank Jonatan Wiik for being a friend and a room-mate as well as for the invaluable help including the preparation of the Swedish version of the abstract.

I am especially grateful to my parents, Lyudmila and Boris, as well as to my sister Svetlana and my nephew Oleg, for their support, patience, understanding and limitless love.

Finally, I would like to express my true thankfulness to my wife, Yuliya, for her care, encouragement and love. Thank you very much for being there for me.

Sergey Ostroumov,
Åbo, May 2015

List of Original Publications

[1] Sergey Ostroumov, Leonidas Tsiopoulos, Marina Waldén, Juha Plosila, Hierarchical agent-based monitoring systems for dynamic reconfiguration in NoC platforms: A formal approach, Advancing Embedded Systems and Real-Time Communications with Emerging Technologies, Ch. 13, IGI Global, pp. 302-333, 2014.

[2] Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Formal Approach to Agent-based Dynamic Reconfiguration in Networks-On-Chip, Journal of Systems Architecture 59(9), Elsevier, pp. 709-728, 2013.

[3] Sergey Ostroumov, Pontus Boström, Marina Waldén, Derivation of Parallel and Resilient Programs from Simulink Models, In Proceedings of International Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE Computer Society Conference Publishing Services (CPS), pp. 416-420, 2015.

[4] Sergey Ostroumov, and Leonidas Tsiopoulos, VHDL Code Generation from Formal Event-B Models, In Proceedings of Digital System Design: Architectures, Methods and Tools, IEEE Computer Society Conference Publishing Services (CPS), pp. 127-134, 2011.

[5] Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Generation of Structural VHDL Code with Library Components From Formal Event-B Models, In Proceedings of Euromicro Conference on Digital System Design, IEEE Conference Publishing Services (CPS), pp. 111-118, 2013.

List of Abbreviations

ASIC	Application Specific Integrated Circuit
BA	Before-after predicate
CTL	Computation Tree Logic
CSP	Communicating Sequential Processes
EPPN	Error-Proof Process Network
ERRIC	Embedded Reliable Reduced Instruction Processor
FIFO	First-In-First-Out
FPGA	Field-Programmable-Gate-Array
H2A	Hierarchical Agent-based Adaptation
HDL	Hardware Description Language
HW	Hardware
NoC	Network-on-Chip
OPS	Operations per Second
PU	Processing unit
RT	Router
SW	Software
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration

List of Figures

List of Tables

Contents

Part I

Research Summary

1 Introduction

The constantly growing demand for high performance along with low power consumption has drawn attention of the research community to parallel computing. However, this shift also increased complexity of systems very dramatically, so that the systems became more prone to various faults. In this chapter, we motivate the research conducted to approach these problems and present the organization of the thesis.

1.1 Motivation

Over the past decades, the Very-Large-Scale-Integration (VLSI) technology scaled down significantly. This allowed placement of more transistors onto a single die and increase of operating frequency in order to provide more performance for sequential programs. However, this has also led to the problem of the power dissipation of a chip. Thus, the increase of chip performance is no longer a matter of incrementing chip frequency, but a matter of finding new ways to satisfy performance demand of modern applications [1]. Moreover, since the number of processing elements constantly increases according to Moore's law, the new technology must be scalable. These challenges have shifted the focus of research to the direction of distributed and parallel computing.

To provide scalability along with parallel computation, a Network-on-Chip (NoC) interconnect paradigm has been proposed [2]. An NoC-based many-core platform can integrate tens or even hundreds of processing units (cores) that communicate with each other. This enables high computational power whilst fulfilling timing constraints and low power consumption. There are also commercially available platforms that utilize NoC, e.g., TilePro by Tilera [3] and Intel Single Cloud Chip [4]. However, due to the high level of on-chip integration, the probability of various faults increases [5]. Moreover, high computational load may cause the creation of hotspots leading to thermal problems [6]. Additionally, transient or intermittent (soft) faults can be caused by radiation [7], which is frequent in space, but becomes an issue also at the ground level [8]. This requires the platforms to be highly resilient to these faults.

Various mechanisms have been explored to achieve resilience to faults. Some of them propose a specific architecture (e.g., [9]). Others suggest replication of the execution tasks (e.g., [8]). While the specific architecture approaches make them difficult to apply to different application domains, the duplication of the execution tasks may significantly reduce utilization and, hence, performance of the underlying platform.

To overcome these disadvantages, agent-based management for many-core platforms has been proposed [10]. Agents allow continuous monitoring of the platform and its dynamic reconfiguration when required. This helps the platform to avoid overloading with management activities while it performs routing of packets, for example. Clearly, a greater number of resources in the platform requires a larger number of agents. Thus, the agents need to be organized into a multi-level hierarchy in order to provide efficient platform management [11].

An agent-based system is typically a composition of software (SW) and hardware (HW) depending on the functionality, timing requirements and complexity of algorithms [12]. Particularly, the higher in the hierarchy the agent is, the more diverse its management activities are. In other words, these agents have more functionality and are more complex than those lower in the hierarchy. Hence, these types of agents are typically implemented as SW which provides a high level of flexibility. On the other hand, the lower level agents have to provide fast and efficient monitoring. Consequently, they are usually implemented as HW [12]. Clearly, the complexity of the agent-based system is high, which increases the risks of introducing design faults during the system development. Inadequate behaviour of the agents may lead to, e.g., improper resource allocation [13] which can cause inadmissible effects. Therefore, rigorous design approaches need to be undertaken.

One of the approaches to tackle design faults is the formal development of a system model. It aims at deriving correct systems by stepwise unfolding of the system functionality through model transformations called refinements and mathematically proving their correctness [14]. In this thesis, we adopt the Event-B formal framework [15]. Event-B supports system level modelling (specification) and allows us to reason about correctness of the model with respect to postulated properties by theorem proving. Moreover, the specification within Event-B follows the refinement approach, where each refinement step is shown to be correct by theorem proving as well. In addition, Event-B has a mature tool support, namely the Rodin platform [16]. The platform is extensible in the form of plug-ins, which allows us to expand the platform functionality. Since code generation is a natural step for formal design flow, there are, e.g., plug-ins for the Rodin platform that allow one to derive code in software languages, e.g., C [17], Java [18], and others [19]. However, due to the fact that hardware description languages (HDLs) differ in semantics and syntax from software languages, the same methods and techniques cannot be directly and completely applied to hardware design and code generation. Thus, this problem also requires attention.

1.2 Organization of thesis

The thesis consists of two parts: Part I that summarizes the research and Part II that presents the original research publications. The remainder of Part I is structured as follows. Section 2 introduces the notions and definitions used throughout the thesis. In Section 3, we identify the research challenges that arise in the efficient and resilient systems design. Section 4 summarizes the original research publications and illustrates how the papers address the postulated research problems. In Section 5, we review the approaches related to the questions posed in the thesis. Finally, Section 6 presents the conclusion and outlines future research directions.

2 Background

This chapter outlines the main definitions and concepts used throughout the thesis. It discusses the need for many-core platforms that provide parallel computations. The chapter also describes the notion of agent-based systems and presents the architecture of a many-core platform augmented with the agent-based management. Finally, it illustrates the languages needed to approach the problems addressed in the thesis.

2.1 Many-core platforms

Network-on-Chip (NoC) [2] has been proposed as a scalable interconnect paradigm for many-core platforms which can provide high computational performance fulfilling timing constraints and low power consumption (Fig. 1). A typical NoC-based scheme consists of tiles. The tiles include processing units (PUs) and routers (RT) [2]. The routers provide communication between the tiles by routing packets utilizing various routing algorithms. Within the scope of this thesis, we assume deterministic routing, which is dead-lock and live-lock free [20] as well as provides low latency and fulfils timing constraints [21]. The routers usually incorporate First-In-First-Out (FIFO) buffers [22][23], so that the flow of data is preserved.

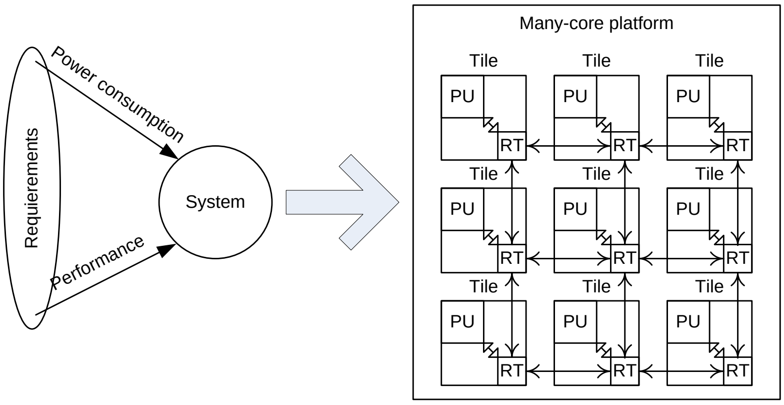


Figure 1: From requirements to many-core platforms

Many-core parallel platforms achieve a high level of performance. For instance, an 80-Tile TeraFLOPS Processor provides more than 1.0E+12 floating point *operations per second* (OPS) [24]. The TilePro64 processor can achieve up to 443E+09 OPS [3].

The advantages of many-core platforms enable their use in various applications, especially in complex critical systems from, for example, biomedical [25] and aerospace domains [9]. However, the use of these platforms in critical domains requires the platforms to be resilient.

2.2 Agent-based management systems

Resilience of a system is defined as the persistence of dependability when facing changes [26]. *Dependability*, on the other hand, is the ability of a system to avoid *failures* that are more frequent and severe than acceptable. Dependability is an integral term that includes availability, reliability, safety, integrity and maintainability [27][28]. A failure is usually a result of an *error* which is caused by a *fault.* A fault is a flaw within a system. It can be, e.g., a design fault (a bug) in software or a physical fault of a hardware component. Faults can be classified as transient, intermittent or permanent [28].

In this thesis, we focus on the reliability attribute of dependability and consider *physical failures* of PUs of a many-core platform. The examples that lead to these failures include: high level of on-chip integration that results in the increase of the probability of various faults [5], high computational load that may cause creation of hotspots leading to thermal problems [6] and radiation which becomes an issue at the ground level and increases probability of transient faults [8].

To provide resilience and maintain performance at an acceptable level, we employ an *agent-based management system* [9][11] (Fig. 3). The term *agent* comes from the artificial intelligence field. One classic definition of an agent is: “An agent is something that perceives and acts resiliently and autonomously” [29]. In the context of many-core platforms, we define an agent more precisely as: “An agent is a piece of software or hardware that acts resiliently and autonomously and enables the platform to achieve some objectives”. An agent-based management system is a system that comprises of a number of agents that communicate with each other [9].

Since a many-core NoC-based platform can integrate tens or even hundreds of tiles, it is reasonable to organize agents into a multi-level hierarchy for efficient and effective platform management [11]. Typically, a three-level hierarchy which we adopt in this thesis is considered efficient [12]. The hierarchy consists of three layers, namely the platform agent, cluster and cell agent layers (Fig. 2). The agents communicate via a dedicated communication mechanism, e.g., a dedicated NoC-based scheme. This allows the platform not to be overloaded with management activities while the platform performs, e.g., routing algorithms.

The platform agent is responsible for the whole platform. It performs the initial mapping of applications, creates, adjusts and destroys cluster and corresponding cluster agents as well as releases resources. The cluster agents manage clusters, i.e., regions (sets of cores), where applications are mapped. The cell agents are local monitors assigned to each tile.

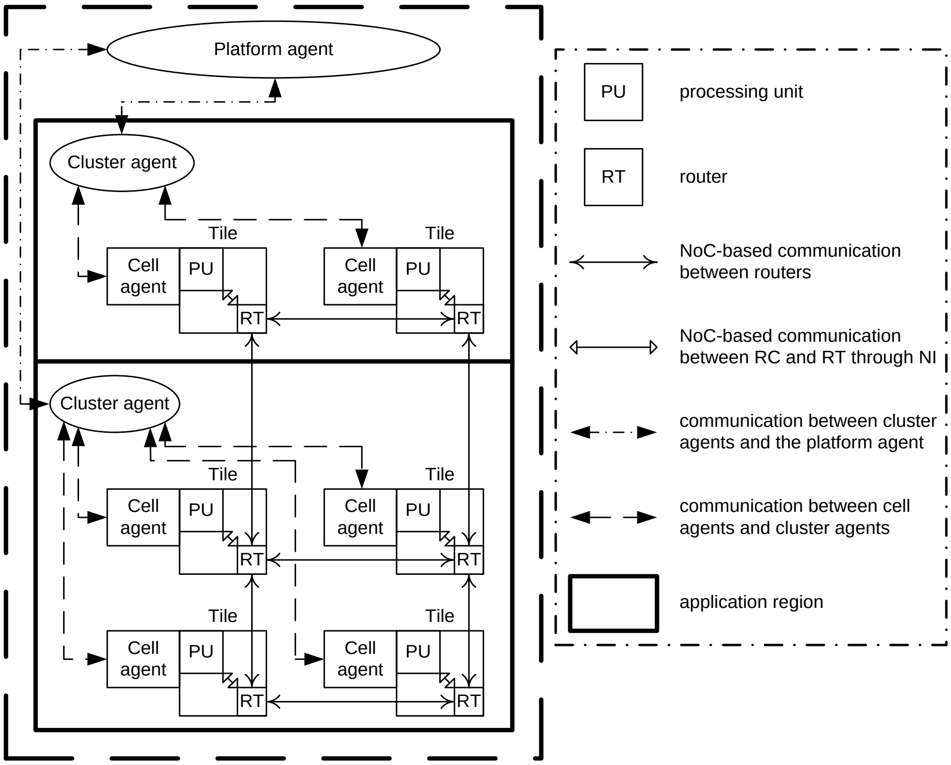

Platform agent
Cluster agent
Tile
Cell agent
PU
RT
Cluster agent
PU
processing unit
RT
router
NoC-based communication between routers
NoC-based communication between RC and RT through NI
communication between cluster agents and the platform agent
communication between cell agents and cluster agents
application region

Figure 2: A many-core platform architecture with hierarchical agents

2.3 Modelling languages

To develop a system, designers commonly utilize various languages. In this section, we describe languages that have been used to approach the problems addressed in the thesis. We start with the description of the formal Event-B framework. We then describe a semi-formal modelling technique widely used in industry, namely Simulink. Finally, we outline a non-formal language likewise widely used in industry, namely VHDL, that enables synthesis and analysis of hardware implementations.

2.3.1 The Event-B formalism

The main formal framework we use in this thesis is the Event-B formalism [15]. There are several advantages this formalism offers. Firstly, it allows us to build system level models. Secondly, it supports the refinement approach such that a model is built top-down in a correct-by-construction manner. Thirdly, the development follows rigorous rules with mathematical proofs of correctness of models. Last but not least, it has a mature tool support extensible in the form of plug-ins, namely the Rodin platform [16]. Let us now describe the structure and notation of Event-B.

A specification in Event-B consists of *contexts* and *machines*. The relationship between them is shown in Fig. 3. A context can be *extended* by another context whilst a machine can be *refined* by another machine. Moreover, a machine can refer to the contents of the context (to "*see*").

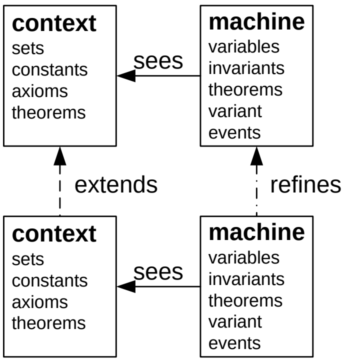


Figure 3: Event-B contexts and machines: contents and relationship [15]

A context specifies static structures such as data types in terms of *sets*, *constants*, properties given as a set of *axioms*. One can also postulate and prove *theorems* that ease proving effort during the model development.

A machine models the behaviour of a system. The machine includes *state variables*, *theorems*, *invariants*, a *variant* and guarded transitions (*events*). The invariants represent constraining predicates that define types of the state variables as well as essential properties of the system. The overall system invariant is defined as the conjunction of these predicates.

A variant is a natural number or a finite set. It is required to show the termination of certain events that can be executed several times in a row, e.g., modelling a loop.

An event describes a transition from a state to a state. The syntax of the event is as follows:

$$E = \textbf{ANY}\ x\ \textbf{WHERE}\ g\ \textbf{THEN}\ a\ \textbf{END}$$

where x is a list of event local variables. The *guard* g stands for a conjunction of predicates over the state variables and the local variables. The *action* a describes a collection of assignments to the state variables.

We can observe that an event models a guarded transition. When the guard g holds, the transition can take place. In case several guards hold simultaneously, any of the enabled transitions can be chosen for execution non-deterministically. If none of the guards holds, there is a deadlock.

When a transition takes place, the action a is performed. The action a is a composition of the assignments to the state variables executed simultaneously and denoted as $||$. An assignment can be either deterministic or non-deterministic. A deterministic assignment is defined as $v := E(w)$, where v is a list of state variables, E is a list of expressions over some set of state variables w. A non-deterministic assignment is specified as $v :| Q(w, v')$, where $Q(w, v')$ is a predicate over some state variables w and a new value v' of variable v. The variable v obtains such a value v' that $Q(w, v')$ holds.

These denotations allow for describing semantics of Event-B in terms of *before-after predicates* (BA) [30]. Essentially, a transition is a BA that represents a relationship between the model state before (v) and after (v') the execution of an event. Hence, the correctness of the model is verified by checking if the events preserve the invariants (INV) and are feasible to execute (FIS) in case the event action is non-deterministic:

$$Inv \land g_e \Rightarrow [BA_e]Inv \qquad \text{(INV)}$$

$$Inv \land g_e \Rightarrow \exists v' \,.\, BA_e \qquad \text{(FIS)}$$

where Inv is a model invariant, g_e and BA_e are the guard and the before-after predicate of the event e, respectively. The expression $[BA_e]Inv$ stands for the substitution in the invariant Inv according to BA_e.

In addition, deadlock freedom of the specification may be corroborated. A deadlock free specification stands for the case where there exists at least one event that can be executed. To achieve this, one needs to postulate a machine theorem that includes the guards of all the events connected with disjunction and show that the proof obligation (DLF) [15] is preserved:

$$\forall S, C, V \,.\, A \wedge I \Rightarrow \bigvee_{i=1}^{n} g_i \qquad \text{(DLF)}$$

where n is the number of events and g_i is the guard of the i-th event. The structures S, C and A represent sets, a collection of constants and axioms introduced into a context, respectively. The structures V and I stand for a set of state variables and a set of invariants of a machine, respectively.

Since the specification development in Event-B follows the refinement approach, one has to prove that the more concrete (refined) events simulate their abstract counterparts. To show this, the refined events must preserve the guard strengthening (GRD) and action simulation (SIM) proof obligations [31] as well:

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \,.\, A \wedge A_r \wedge I \wedge I_r \wedge g_r \Rightarrow g \qquad \text{(GRD)}$$

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \,.\, A \wedge A_r \wedge I \wedge I_r \wedge BA_{er} \Rightarrow BA_e \qquad \text{(SIM)}$$

where all letters with subscript “r” stand for the refined versions of the aforementioned structures.

To prove that new events executed several times in a row terminate, one also has to show that these events are consistent with a variant. In particular, these events have to preserve either of the following proof obligations depending on whether the variant is a natural number (VAR_N) or a finite set (VAR_S) [31]:

$$\forall S, C, V \,.\, A \wedge I \Rightarrow Var \in \mathbb{N} \wedge [BA_e]Var < Var \qquad \text{(VAR\_N)}$$

$$\forall S, C, V \,.\, A \wedge I \Rightarrow finite(Var) \wedge card([BA_e]Var) < card(Var) \qquad \text{(VAR\_S)}$$

where Var is a variant that denotes a numeric expression or a finite set of values. The expressions $finite(Var)$ and $card(Var)$ specify finitness and cardinality of the set variant, respectively.

In case the model needs to be deadlock free, one can show the relative deadlock freedom, i.e., all concrete events should not deadlock more frequently than the abstract ones. Therefore, the disjunction of the abstract guards should imply the disjunction of the concrete guards (proof obligation (DLFR)) [15]:

$$\forall S, C, V \,.\, A \wedge I \wedge I_r \wedge \bigvee_{i=1}^{n} g_i \Rightarrow \bigvee_{j=1}^{m} g_{rj} \qquad \text{(DLFR)}$$

where m is the number of concrete events and g_j is the guard of the j-th event.

The Rodin platform [16], a tool support for Event-B, automatically generates and attempts to discharge (prove) the necessary proof obligations. The best practices encompass the development of the specification in such a manner that 90-95% of the proof obligations are discharged automatically. However, the tool sometimes requires the user assistance provided via the interactive prover. Typically, the claims that are difficult for the automatic prover to discharge require case distinction and/or data substitution.

2.3.2 Simulink dataflow diagrams

The control logic of a system can also be modelled by using semi-formal techniques. One such a technique which is widely used in industry is Simulink [32]. A Simulink model is a hierarchical dataflow diagram that describes the essential functionality of a system by hiding implementation details. The model consists of a collection of *functional blocks* that have *in-ports* (inputs) and *out-ports* (outputs) allowing connections between blocks via *typed signals*. The blocks may have *parameters* that are initialized at the beginning of the execution and remain constant during the execution. Moreover, the blocks can contain *memory*, in which case the output value depends not only on the inputs, but also on the previously computed value.

The hierarchical diagrams are achieved by grouping blocks into *sub-systems*. There are two types of sub-systems in Simulink: *virtual* and *atomic* [33]. Virtual sub-systems are used for the structural purpose only and do not affect the model execution. They can be seen as containers for functional blocks that are expanded by the Simulink engine in place before execution. Atomic sub-systems are treated as single atomic units.

Fig. 4 illustrates an example of a Simulink model. The model in Fig. 4, a) contains two in-ports and one out-port. It includes a constant parameter as well as memory. This model is grouped into a sub-system presented in Fig. 4, b).

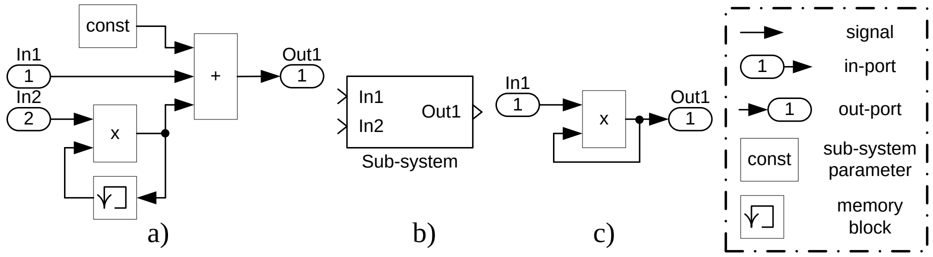


Figure 4: Simulink models: a) sub-system content, b) sub-system block, c) algebraic loop

The models can be *continuous* or *discrete*. We consider discrete-time models with atomic sub-systems that specify periodic real-time systems. Each block in a discrete-time model is evaluated at regular intervals with a specified sampling period. We further assume that the model is single-rate, i.e., all its sub-systems fire at the same time intervals. Additionally, we assume causal models, where the output of a block has no direct connection to the input of the same block. The direct connection of an output to an input is also known as an algebraic loop [34] (Fig. 4, c)).

2.3.3 Hardware description language VHDL

Once the model of a system is derived, one can carry out the performance analysis by translating the model into an implementation in a high-level programming language. For instance, one can generate C [16], Java [18] or other programming language [19] code from an Event-B model. The Simulink design environment supports the generation of C code as well [35]. However, a complex system such as an agent-based management system is often a composition of two parts: software and hardware [12], where hardware can be specified using a hardware description language (HDL). Clearly, HDLs differ in semantics and syntax from software programming languages, which makes it difficult to apply the software code generation techniques to hardware design and code generation. Thus, the hardware part requires attention.

We have chosen VHSIC Hardware Description Language (VHDL) as the target language. This language is standardized [36] and widely used in industrial hardware design. Moreover, there are a number of tools that support VHDL, e.g., Quartus-II by Altera [37]. These tools can synthesize the description and provide information about area consumption and performance. Let us now describe the essential parts of a VHDL description.

There are two basic elements a VHDL description consists of: an entity and an architecture (Fig. 5). The entity defines the interface of a hardware component whilst the architecture specifies its behaviour.

Every entity must have a name and ports. The entity can have parameters defined in the *generic* clause. The interface ports contain input and output signals whose type and direction have to be specified explicitly. The example entity in Fig. 5 has a parameter whose type is a natural number and initial value equals 10. Moreover, it has two input ports and one output port differentiated by the keywords **in** and **out**, respectively. The types of the ports are bit vectors.

The entity can be assigned with an *architecture* that implements the behaviour of the hardware component. The architecture consists of 2 parts: declaration and body. The declarative part includes definitions of internal signals as well as interfaces of other hardware components. The body specifies the function of the hardware component, where a designer can instantiate declared components by using the keywords *generic map* and *port map*. Moreover, the body can have a *process* that reacts on certain signals introduced into a *sensitivity list* and allows for introducing sequential statements such as ***if*** *(condition)* ***then*** *action* ***end if***. The action in the **if** ... **end if** statement is an assignment of a value to a signal in the form of s <= E, where s is a signal and E is an expression. Every assignment in the process is not instant, i.e., the signals are updated when the whole process completes its execution. Hence, all the signals involved in the assignment are updated simultaneously.

```
entity Entity is
generic ( parameter : natural := 10 );
port ( input1 : in std_logic_vector(parameter-1 downto 0);
         input2 : in std_logic_vector(parameter-1 downto 0);
         output1 : out std_logic_vector(parameter-1 downto 0) );
end Entity;

architecture arch of Entity is

  signal internal_signal : std_logic := ‘0’;
  signal internal_signal_add : std_logic_vector(parameter-1 downto 0);

  component component_add
  generic ( width : natural );
  port ( dataa : in std_logic_vector(width-1 downto 0); datab : in std_logic_vector(width-1 downto 0);
           result : out std_logic_vector(width-1 downto 0) );
  end component;

begin

  add_0 : component_add
  generic map( width => parameter )
  port map( dataa => input1, datab => input2, result => internal_signal_add );

  process_add :
  process ( input1, input2, internal_signal, internal_signal_add ) is begin
    if (internal_signal = ‘0’) then internal_signal <= ‘1’; end if;
    if (internal_signal = ‘1’) then internal_signal <= ‘0’; output1 <= internal_signal_add; end if;
  end process;

end arch;
```

Figure 5: A VHDL description

The declaration part of the example architecture in Fig. 5 contains two internal signals and the hardware component component_add. The body instantiates the declared component by specifying the width and mapping inputs of the component to the input ports and the result to one of the internal signals. The body also contains a process. The process reacts on changes of the input ports as well as internal signals. Upon reaction, the process updates the output of the entity with the result computed by the component component_add. Essentially, the architecture implements a simple adder.

3 Research objectives

The main objective of the thesis is to determine methodological aspects of the design and implementation of agent-based systems for many-core platforms. We motivate the research and postulate research questions addressed in the thesis.

During the design, we simultaneously consider performance and resilience aspects of the platform as well as of the agent-based system. Resilience of the platform is attained by utilizing dynamic reconfiguration of the platform performed by the agents. To achieve resilience of the agent-based system, we employ the rigorous correct-by-construction development. Then, we translate the derived rigorous specification into a programming language in order to evaluate efficiency in terms of performance and area overhead. In addition, we outline the problem of data loss when a many-core platform is dynamically reconfigured.

3.1 Rigours design of agent-based management system

In the previous section, we have shown that many-core platforms are envisaged to be used in complex critical systems, which requires the platforms to be resilient to faults. To achieve resilience of the platform, we employ a hierarchical agent-based management system embedded in the platform. However, the agents are dynamic autonomous entities that have be to resilient as well. Their inadequate behaviour may lead to undesirable consequences. An unpredictable behaviour of an agent may cause problems related to, e.g., resource allocation [12]. Furthermore, a large number of resources provided by many-core platforms requires the hierarchical organization of agents, which is also needs to be taken into account. Hence, formal methods are required to ensure correctness of the agents and their reliable behaviour. This leads us to the first research question:

RQ1: *How to take into account hierarchical organization of agents? Moreover, how to rigorously design a hierarchical agent-based management system such that its behaviour can be trusted?*

3.2 Integration of dynamic reconfiguration

Once the main steps are derived, resilience of the underlying platform can be considered. A common approach to achieve resilience to faults is to use redundancy such that replicas of an application are run in parallel with the main execution (see e.g., [5][38]). Clearly, the use of replicas may reduce the utilization of the platform, decrease its performance and increase power consumption. Thus, these attributes are orthogonal (Fig. 6).

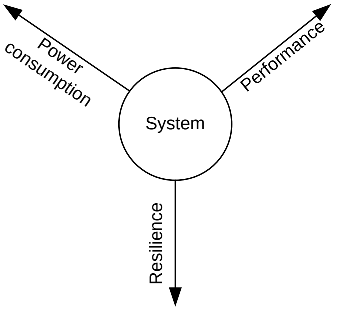


Figure 6: Orthogonality of performance and resilience

To overcome these limitations, dynamic reconfiguration needs to be undertaken. Dynamic reconfiguration includes dynamic voltage and frequency scaling [12], task migration (reallocation) [5][39] as well as partial reconfiguration of FPGA-like regions [40][41] techniques. Generally, the aim of these techniques lies in providing a better balance between, e.g., power consumption and performance. However, they can also be used to achieve resilience. For instance, the reallocation of tasks from failed PUs to some free non-failed ones allows the tasks to continue execution without interruption [5][39]. Hence, the second research question is:

RQ2: *How to integrate dynamic reconfiguration of the platform into agents hierarchy, so that an acceptable level of performance is maintained? Additionally, how to show that the agents will behave resiliently under these circumstances?*

3.3 Data loss avoidance

Dynamic reconfiguration is a powerful mechanism to provide resilience and maintain performance. However, the application tasks may lose data when the platform is reconfigured (see, e.g., [42]).

When a control task runs, it executes the three main operations: reading input data (either from the environment or from packets), processing the received data (i.e., executing a function) and sending the processed data further (either to other tasks or to the environment). The fault occurrence within these operations is captured by the following fault scenarios:

(1) *a fault occurs before a task reads any input data*.
(2) *a fault occurs while a task reads input data*.
(3) *a fault occurs before the task sends the processed data*.
(4) *a fault occurs while a task sends data*.

In case (1), a task can still read the input data after reallocation as they remain intact. In case (2), the task reads packets from some queues, but fails to read from others. Thus, some pieces of data may be lost. In case (3), the task has read all the input data, but has not finished processing them or has not been able to send the processed data. Hence, the task loses data of one firing. Finally, in case (4), some successor tasks may receive packets with the new data while others may not. This can lead to the desynchronized data reception by the successor tasks. Consequently, some data are lost.

We can observe that the fault occurrence may lead to the data loss depending on the point when a fault occurs. The loss of data may affect the production of the correct output result. This raises our third research question:

RQ3: *How to avoid data loss when the many-core platform is reconfigured?*

3.4 Performance evaluation

The formal specification of a system guarantees its correctness qualitatively, i.e., proper functional behaviour with respect to the postulated properties. However, non-functional attributes such as performance and overhead should be evaluated quantitatively. To achieve this, the derived specification needs to be translated into a high level programming language. This leads to our fourth research question:

RQ4: *How to evaluate performance of the derived agent-based system? Specifically, how to translate a formal model into a synthesizable code?*

3.5 Research methods

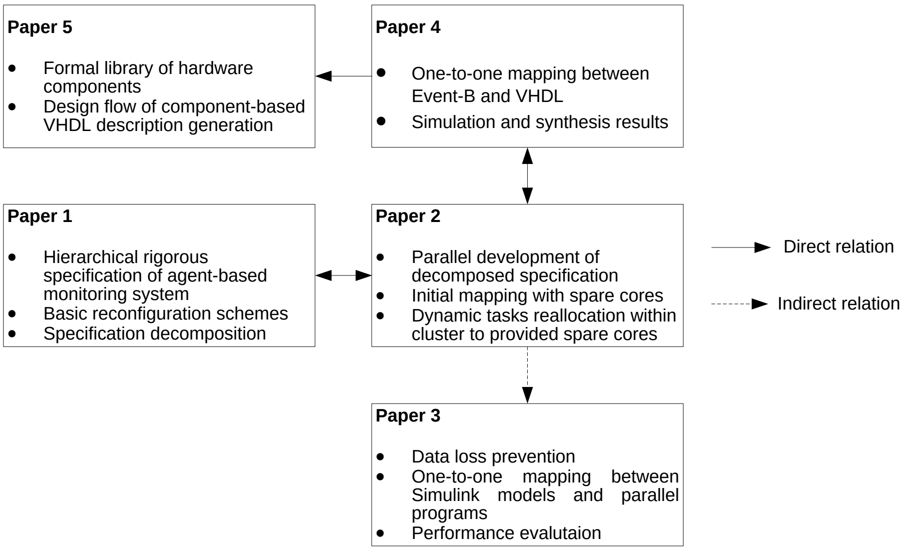
We approach the first and the second research questions by analysing the functionality of the agent-based system and possible reconfiguration schemes applicable to the platform (e.g., task reallocation, hardware reconfiguration). We then utilize the Event-B formalism to develop the specification of the system in a hierarchical and correct-by-construction manner. Event-B allows us to ensure the correctness (i.e., trusted behaviour) of the agents with respect to postulated properties using a proof-based development process.

To tackle the third research question, we analyse the outlined fault scenarios. Using the features of a many-core platform, we then develop algorithms to prevent tasks from data loss when they are reallocated.

Finally, we explore the fourth research question by using semi-formal and informal techniques. We use the derived specification of the agent-based system as the base to obtain the implementation of the system through the automated code generation. The synthesis of the generated code allows for efficiency evaluation in terms of area overhead and performance. In addition, we deploy the proposed algorithms on a commercially available many-core platform, namely TilePro by Tilera [3], and evaluate communication and computation performance of the tasks.

4 Overview of Research Publications

The main research results are documented as the peer-reviewed papers given in Part II of the thesis. Fig. 7 illustrates the relationship between the publications, where the solid arrows depict a direct relation whilst the dashed arrow represents an indirect relation between the papers.



Paper 5
Formal library of hardware components
Design flow of component-based VHDL description generation
Paper 4
One-to-one mapping between Event-B and VHDL
Simulation and synthesis results
Paper 1
Hierarchical rigorous specification of agent-based monitoring system
Basic reconfiguration schemes
Specification decomposition
Paper 2
Parallel development of decomposed specification
Initial mapping with spare cores
Dynamic tasks reallocation within cluster to provided spare cores
Direct relation
Indirect relation
Paper 3
Data loss prevention
One-to-one mapping between Simulink models and parallel programs
Performance evalutaion

Figure 7: Relation between research publications

This chapter overviews the contents of these research publications and highlights the contribution of each paper. In addition, it indicates the contribution of the author.

Paper 1

Sergey Ostroumov, Leonidas Tsiopoulos, Marina Waldén, Juha Plosila, Hierarchical agent-based monitoring systems for dynamic reconfiguration in NoC platforms: A formal approach, Advancing Embedded Systems and Real-Time Communications with Emerging Technologies, Ch. 13, IGI Global, pp. 302-333, 2014.

This paper addresses the first and partially the second research questions. It describes the main steps of the development process of a three level agent-based system for a many-core 2D mesh Network-On-Chip platform. Particularly, we specify an arbitrary platform and show the process of introducing each level of the agents hierarchy through correctness preserving model transformations – refinements – using the Event-B formalism [15]. We consider platform, cluster and cell level agents. We also present possible variations of the platform reconfiguration and integrate them into the formal model. These reconfiguration schemes include:

1) task reallocation and application remapping performed by the platform agent,
2) dynamic voltage and frequency scaling executed by the cluster agents and
3) local reconfiguration of the platform cores performed by the cell agents.

Author's contribution: The idea originated from the co-authors of the paper. The main responsibility of the author was the development of the formal specification. Additionally, the author was responsible for the paper.

Paper 2

Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Formal Approach to Agent-based Dynamic Reconfiguration in Networks-On-Chip, Journal of Systems Architecture 59(9), Elsevier, pp. 709-728, 2013.

This paper addresses partially the first and completely the second research questions. In this paper, we continue the rigorous development of the agent-based management system considering requirements on efficiency. Specifically, we propose to allocate a number of spare cores within a region for each application being mapped. The number of the spare cores is computed as a half of the number of the required cores. These spare cores are initially allocated on the right side of the application region. This initial configuration (mapping) is performed by the platform agent. In case a fault occurs in cores within the region, a corresponding cluster agent is delegated to utilize the allocated spare cores. It reallocates a task from a faulty core to a spare one in accordance with the algorithm proposed in the paper. Then, a cell agent can initiate the local reconfiguration procedure. Therefore, the functionality of configuring and reconfiguring the platform is evenly distributed among the agents. This allows for efficient performance of the agents as well as the many-core platform.

Author's contribution: The author was responsible for the formal development, simulations and writing the core of the publication.

Paper 3

Sergey Ostroumov, Pontus Boström, Marina Waldén, Derivation of Parallel and Resilient Programs from Simulink Models, In Proceedings of

International Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE Computer Society Conference Publishing Services (CPS), pp. 416-420, 2015.

This paper addresses completely the third and partially the fourth research questions. In this paper, we present an approach to generation of a parallel C code from a discrete single-rate Simulink model that specifies periodic control logic. Relying on this, we propose a mechanism, where the tasks can continue execution without data loss. The paper includes performance evaluation without and with the proposed mechanism using an industrial case study. The evaluation results are obtained using a commercially available platform TilePro [3]. They show that the proposed approach decreases performance of an application by only about 1% while allowing it to produce the expected result, i.e., to satisfy resilience requirements.

Author's contribution: The work was initiated by the author. Moreover, the author was responsible for the implementation and evaluation of the proposed mechanism. Additionally, the author was responsible for the publication.

Paper 4

Sergey Ostroumov, and Leonidas Tsiopoulos, VHDL Code Generation from Formal Event-B Models, In Euromicro Conference on Digital System Design: Architectures, Methods and Tools, IEEE Computer Society Conference Publishing Services (CPS), pp. 127-134, 2011.

This paper partially addresses the fourth research question. In this paper, we study a one-to-one mapping between the Event-B formalism and VHSIC hardware description language (VHDL) in order to analyze area overhead and performance. The mapping is based on the similarities in the structures of a formal model and a VHDL description. Additionally, we show algorithmic steps required to derive a synthesizable VHDL implementation from a formal model. These steps are implemented in the form of a plug-in to the Rodin platform that supports the Event-B formalism. The correctness of the code generation is shown through the stepwise comparison of simulation results for the model and the code. To support the approach, we present the development of a simplified version of an industrial case study developed in a stepwise refinement manner and code generation for it. In addition, we illustrate synthesis results that illustrate performance and area occupied by the generated VHDL description.

Author's contribution: The author provided a case study and developed its formal specification. The author was also responsible for writing the main parts of the publication and for the development of the tool support.

Paper 5

Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Generation of Structural VHDL Code with Library Components From Formal Event-B Models, In 16th Euromicro Conference on Digital System Design, IEEE Conference Publishing Services (CPS), pp. 111-118, 2013.

This paper contributes to the fourth research question. Due to strict requirements, a VHDL description generated as mentioned in the previously described paper may not be sufficient. It may be crucial when the agents need to react rapidly due to the highly dynamic nature of the applications and the many-core platform. Hence, in this paper, we propose a method for deriving a structural (i.e., component-based) description from a formal model. We develop a formal library of hardware library components which allows designers to generate a component-based description. We show that a structural description obtained from a formal model following the proposed method requires less area and performs better than a non-structural one. In addition, we present a design flow that follows the usual refinement-based development and ends in an automated code generation.

Author's contribution: The author was responsible for the development of the formal library and for the implementation of the tool support. Additionally, the author was responsible for the publication.

Summary

These publications address the research challenges postulated in the previous section. Tab. 1 summarises the research publications and the research questions that have been addressed by each publication. The contributions of the thesis can be summarized as follows:

1. Formal rigorous development of agent-based systems taking into account their hierarchical organization
2. Various dynamic reconfiguration procedures integrated into the agents functionality simultaneously considering efficiency (performance, overhead etc.) and resilience attributes
3. A mechanism that prevents data loss when the underlying platform is reconfigured at run-time

4. An approach to model translation into a synthesizable description in order to facilitate easier derivation of the implementation and evaluation of non-functional properties in a real-world environment

Table 1: Summary of research questions and publications

Research question		**Paper(s)**
RQ1	a) How to take into account hierarchical organization of agents? b) Moreover, how to rigorously design an agent-based management system such that its behaviour can be trusted?	1 1,2
RQ2	How to integrate dynamic reconfiguration of the platform into agents hierarchy, so that an acceptable level of performance is maintained? Additionally, how to show that the agents will behave resiliently under these circumstances?	1,2
RQ3	How to avoid data loss when the many-core platform is reconfigured?	3
RQ4	a) How to evaluate performance of the derived agent-based system? b) Specifically, how to translate a formal model into a synthesizable code?	3,4 4,5

5 Related work

In this chapter, we review the approaches related to the research questions posed in this thesis. We first discuss the approaches focusing on the development of efficient and resilient many-core platforms. We then analyze the research conducted towards design of agent-based systems in an informal and a formal manner. Finally, we conclude the related work with the approaches that focus on the translation of the rigorous specifications into synthesizable descriptions.

5.1 Resilience and Many-Core Platforms

Motamedi et al. [9] have proposed a fault-tolerant reconfigurable NoC considering application specific architecture for avionic systems. Particularly, they use a star network topology as the main active formation where the cockpit switch is placed in the centre of the topology. The redundancy is achieved by placing redundant links in the system. When a fault is detected, the topology is switched (reconfigured) from the star formation to the ring one. Additionally, the authors utilize the Embedded Reliable Reduced Instruction Processor (ERRIC) as a computational unit. The instruction set of ERRIC has been specially designed to tolerate malfunctions caused by permanent faults. Using the prototyping results, the authors illustrate that the overhead of their approach is marginal while the required level of fault-tolerance is achieved. Although ERRIC is used as a computational unit, it has a reduced instruction set which may not be applicable to application domains other than avionics. Moreover, ERRIC is implemented on Field-Programmable-Gate-Array (FPGA) or Application-Specific-Integrated-Circuit (ASIC), where a physical fault may also occur and, hence, this processing unit may not be operable any more.

Instead of focusing on the topology reconfiguration, we focus on faults of the processing units. When a fault is detected, the agents execute various reconfiguration procedures considering performance of the applications. These procedures allow for executing applications without interruption and enable functional recovery of the platform. We consider a topologically fixed NoC-based platform which is not application specific. Although we utilize a specific topology, our approach is applicable to other topologies and other types of the routing schemes as it does not depend on them. Nevertheless, redundant routers (and/or links) can complement our approach.

An approach to remapping with spare cores has been proposed by Chou and Marculescu [39]. The authors study three possible schemes of spare cores allocation at the system level only, without considering assignments of spare cores within application/cluster regions. The three possible assignments include: 1) side assignment, 2) random assignment and 3) uniform assignment. The authors provide the metrics for evaluation of the task remapping to spare cores and point out that the remapping to the randomly placed spare cores performs better than to the spare cores placed to the side of the system. Clearly, a spare core allocated at a great distance from an application drastically decreases performance of the entire system.

In contrast, we propose to incorporate spare cores at the side of each cluster (region) instead of spare cores assignment at the system level only. Depending on the size of an application, a fixed number of spare cores is provided to a corresponding cluster agent allowing it to tolerate faults while maintaining the performance of the computations at an adequate level. We provide an algorithm for spare cores utilization at the cluster level. In addition, we propose to initiate a local reconfiguration procedure on a faulty cell in order to recover its functionality. When this procedure is complete, the cluster agent reallocates the task back restoring the original performance of computations.

There are several other works that propose dynamic (re)mapping of applications. Some of them are single-objective, i.e., they focus on minimizing, for instance, energy consumption [43]. Other works address simultaneous optimization of mapping and software-hardware partitioning without considering faults of the platform [44]. In our approach, we propose to integrate and uniformly distribute the reallocation and reconfiguration functionality within the agents hierarchy such that a high level of fault-tolerance is achieved while performance remains at an adequate level. Furthermore, to the best of our knowledge, all of these approaches have been developed informally, w.r.t. correct-by-construction and proof-based development, while our approach is supported by the Event-B formal framework which provides the development of a system through refinements and correctness proofs.

One technique to provide resilience to physical faults is to use redundancy. For example, Bolchini, Carminati and Miele [8] propose to replicate the whole application or some of its threads in order to detect and tolerate failures of processors. They assume data parallel programs as well as consider duplication with comparison, triplication and duplication with comparison and re-execution fault-tolerance (FT) techniques. The authors propose an adaptation engine that acts according to the evolving environment. They consider several parameters, called knobs, which the adaptation engine needs to take into account. The adaptation engine incorporates observe-decide-act loop that allows for achieving adaptability.

Another approach to replicating dataflow actors has been proposed by Pinello, Carloni and Sangiovanni-Vincentelli [38]. The authors consider a fault model, in which components are fail-silent, i.e., they either produce a correct result or produce no result. To effectively detect failures, the authors rely on failure patterns proposed in [45]. These patterns describe a set of vertices of a process graph that may fail within the same iteration. The authors use software replication for critical tasks statically at design time, where each replica is then executed on a separate control unit. Using this technique, the authors describe a fault-tolerant data flow.

An approach to tackle hardware failures in process networks has been proposed by Ceponis, Kazanavicius and Mikuckas [42]. The authors present an extension of Kahn process networks, namely Error-Proof Process Network (EPPN). They give operational semantics of EPPN in the form of labelled transition system, where concurrent nodes communicate via first-in-first-out (FIFO) channels. The nodes can check whether the channels are full or empty and can proceed to blocking write or read, respectively. Relying on this, the authors show a dynamic reconfiguration mechanism where the nodes adapt to faults by transferring actions of a faulty node to an adjacent non-faulty functional node and by accordingly adjusting communication using checks on the FIFO channels. While this mechanism enables further execution of the nodes and helps them to synchronize data, the network may become non-deterministic. When functionality of a failed node is delegated to a non-faulty operating node, data loss occurs. Moreover, this can also lead to deadlocks due to blocking reading and writing. To tackle these problems, the authors introduce the default value. Although the mechanism seems to fulfil continuous and on-time result delivery, the default value may not completely compensate data loss.

Similarly as in [8][38][42], we consider hardware failures of PUs in the underlying many-core platform. However, in contrast to [8][38][42], we rely on dynamic reconfiguration of the platform that can be performed by agents integrated into the platform [9][12]. The dynamic reconfiguration includes tasks reallocation, which enables uninterruptable execution of applications [5][39] and avoids resource wasting caused by duplicating applications or threads (actors). Nonetheless, as in [42], the tasks may lose data when reallocated. To avoid this, we propose an FT mechanism, in which the reallocated tasks operate on the current values instead of the default ones. Therefore, the determinism of the application is preserved. Furthermore, our FT mechanism is not restricted to data parallel applications, but can also be applied to functionally parallel ones.

From the related work above, we can observe that the proposed approaches either are very specific, which makes it difficult to apply them to other application domains, or they do not provide the mechanism for efficient utilization of spare resources, or they focus on specific objectives, so that some attributes are neglected, or they require duplication of the tasks execution, i.e., reduce utilization and performance of the underlying platform. To overcome these drawbacks, agent-based management systems have been studied.

5.2 Formal and Informal Agent-based Systems Design

An informal design of the three-level hierarchical agent-based management system has been explored by Guang [12]. The author studies a design paradigm, namely Hierarchical Agent-based Adaptation (H2A), that addresses the monitoring, decision making and reconfiguration processes. The main objective of the proposed approach is the dynamic performance optimization based on the monitored status. The work presents the hierarchical partition of the functionality among the agents such that monitoring and reconfiguration of a system can be performed efficiently. The author formulates the software/hardware (SW/HW) co-synthesis guidelines for each level of the hierarchy and implements the proposed system in order to evaluate energy consumption and overhead. The evaluation results show that H2A can provide adaptation services to reduce energy consumption while the overhead of the proposed system is marginal. Moreover, relying on the trade-off between energy consumption, latency and area overhead, the author suggests that a separate physical network of agents best fits the aforementioned criteria.

In addition to the energy management, the work in [12] presents an approach to dynamic clusterization in order to address the dependability attribute of the system. The dynamic clusterization allows a cluster agent to be assigned to any cell whilst any cell can be allocated to any cluster. The author considers failures of processing cores assuming that the cell agents can detect them by using various mechanisms. If a core where a cluster agent has been allocated fails, a new core can be configured as a cluster agent. If a core with an application task fails, a spare core can be used as a substitution. Thus, a number of spare cores is needed. However, the decision on the use of spare cores is undertaken by the platform agent. In case there are many applications mapped to the platform, there may not be cores available as spares. In this case, the platform agent may restructure clusters such that a core from one cluster is used as a spare for another cluster. Additionally, when reconfiguring the platform, the platform agent updates the necessary data structures of all the agents such that a new configuration can proceed normally. Hence, the overall performance of the platform may decrease significantly. In the worst case scenario, an application may stop execution and may not be able to produce the result due to various delays and tasks/cores reallocation.

Similarly to [12], we adopt the three-level hierarchical formation of agents that have a physically separated communication network. We also consider performance and resilience requirements simultaneously. However, in contrast to [12], we propose to evenly distribute decision making and reconfiguration activities among the agent levels in the hierarchy in order to achieve the required level of performance and resilience of the platform. In particular, the platform agent maps an application in such a manner that a number of spare cores is provided to the cluster agent during the mapping. Furthermore, the platform agent dynamically creates and destroys a corresponding cluster agent when an application is mapped to and released from the platform, respectively. Additionally, the platform agent can remap the whole application or a particular task in case all the spare cores within the cluster have been utilized and there is a new fault. In this case, the platform agent dynamically adjusts the cluster and the cluster agent according to the new configuration.

Since the platform may contain a large number of cores (thousand-core) and many applications can be mapped on such a platform, the reallocation of a task has to be performed efficiently while still allowing efficient execution of an application. Hence, we integrate dynamic tasks reallocation procedure into the functionality of a cluster agent. The cluster agent manages a set of cores on a smaller scale than the platform agent which is more effective and efficient. We propose an algorithm for efficient utilization of the provided spare cores and evaluate its performance. In our opinion, the three-level architecture we propose provides scalability and coherent structure for many-core NoC-based platforms. The functionality of the agents in this architecture is more balanced enhancing dependability of the platform and not overloading the platform agent. Furthermore, our approach has been developed following the refinement-based and correct-by-construction approach allowing formal verification by discharging proof obligations.

A formal approach to specifying agent-based systems is presented by Andres, Molinero and Nuez [46]. This approach allows designers to describe an agent-based system in terms of communication cellules that are organised into a hierarchy. The authors focus on a mathematical framework for describing such a generic hierarchical agent-based system. However, as the authors state, this approach is difficult to apply for complex systems. Furthermore, this approach does not support reconfiguration procedures nor provides verification means for proving the correctness of the system being modelled. Instead, the approach supports simulation of a system.

The integration of Z notation and X-machines proposed by Ali and Zafar [47] enables modelling of agent-based system behaviour and supports data modelling as well as property analysis. The authors focus on the development of specifications using X-machines and proving their properties using the Z notation. However, the authors do not consider a hierarchical scheme of an agent-based system within their framework, which may lead to increased complexity in its application to large-scale many-core platforms.

In another Event-B approach presented by Lanoix [48], the author refers to a platoon problem, where several vehicles are moving one after another simultaneously. The author considers the vehicles as a situated multi-agent system where agents exchange the information at one level, i.e., a system with a flat architecture. Hence, this approach may not be applicable to hierarchical agent-based systems nor provide scalability for such complex systems. Moreover, the author does not consider faults that may occur in the system and, consequently, reconfiguration procedures to be integrated.

There are also other approaches to the design of agents. For instance, Araragi et al. [49] analyze the three formal methods for modelling agents computations, namely Erdös [50], Nepi^2 [12] and I/O automata [51]. The authors study models of client-request components and present both advantages and drawbacks of these formalisms. In particular, Erdös is an agent programming language suitable for knowledge-based programming and reasoning. The semantics of the programs written in this language can be easily understood due to the knowledge-based style. The programs can also be verified in an automated manner. However, the verification is performed using computational tree logic (CTL) model checking, i.e., it can be only executed for finite state systems. Nepi^2 is a network programming system. It is based on π-calculus, thus, a system can be specified concisely using the π-calculus primitives. However, the Nepi^2 system does not support property specification and verification. Finally, I/O automata allow for modelling components that interact. These automata support compositional, invariant and simulation proofs. However, there is little work in application of these automata to modelling of dynamic systems such as agent-based systems. That is, it may be difficult to apply them to modelling such systems, especially when considering hierarchical organization of agents.

The related work described above shows that the informal development (i.e., implementation in a programming language) of complex agent-based management systems enables designers to quantitatively evaluate non-functional properties, e.g., performance, area overhead and power consumption. On the other hand, rigorous specification of such systems is needed to guarantee their predictable and correct behaviour with respect to functional requirements. To enable reasoning about both functional and non-functional properties, the derived rigorous specification is needed to be translated into a programming language in an automated manner. Since we focus on the HW part of the agent-based management system, there are several works related to HW code generation from formal models.

5.3 Code Generation

Seceleanu [52] proposed an approach to deriving synchronous hardware systems. The approach relies on the Action Systems formalism and enables modeling of a synchronous system as read/write operations. The main idea of the approach is that a combinational (asynchronous) circuit that consists of logic gates is followed by a synchronous component, namely a D-flip-flop, which operates on the clock signal. In addition, the author points out the mapping of such modeling to a behavioral VHDL description, where all operations are at one level of code, i.e., the description without components. Despite the fact that the Action Systems framework is similar to the Event-B formalism, it has a different underlying structure, which makes it infeasible to completely apply this approach to Event-B models. Furthermore, in contrast to this approach, we also propose to derive component-based models and generate structural VHDL descriptions with library components.

Hallerstede and Zimmermann [53] proposed an approach to VHDL code generation from formal B models. The authors describe the mapping between B models and VHDL code through a middleware language B0, which allows one to generate code without components. This approach is adopted by AtelierB tool and supported by industrial partners [54]. Since Event-B is a descendant of the B method that allows us to model reactive systems and has a different underlying structure, the application of this approach to Event-B models is not straightforward. Moreover, we also consider a component-based design flow, where components are injected into a formal model in the form of functions.

This design flow allows for generating a structural VHDL description from an Event-B model.

There also exist several formalisms specifically developed for specification and verification of hardware systems, e.g., Signal [55], Esterel [56] and ForSyDe [57]. Signal is dedicated to data-flow applications domain while Esterel is for control-flow ones. ForSyDe represents the design methodology targeting at covering both domains. The commonality of these languages is that they are all based on the perfect synchrony hypothesis. This hypothesis assumes a zero delay between consuming inputs and producing outputs. In addition, only Signal and ForSyDe support the notion of refinement. Refinement in Signal relies on checking if simulation of inputs and outputs preserves flow-equivalence (model checking) [58]. Refinement in ForSyDe stands for the mapping one process network onto another one restricting these networks to have the same inputs and outputs [57]. Moreover, these transformations have to be performed according to the predefined library.

BlueSpec [59] has been proposed as another solution to formal hardware verification and code generation. The language represents an extension of SystemVerilog and has a sound semantics allowing one to verify certain properties. It also supports design by refinement offering a possibility of integrating automated reasoning into the design flow [60]. However, automated verification of system correctness is provided by external theorem provers or model checkers such as PVS [60] and SPIN [61].

In contrast to these approaches, we propose to use the Event-B formalism, which provides data and superposition refinement [62]. These types of refinement allow for stepwise unfolding of system functionality without restricting the model to have the same number of variables in refinements. Furthermore, one can postulate vital properties in terms of invariants for every refinement step. Following this approach, the discharging (proving) of proof obligations serves as the guarantee that each refinement step preserves invariants and that concrete refinement step sustains their abstract counterparts. After the required model is derived and proved correct, a behavioral or structural VHDL description can be generated directly from the model.

Evans [63] describes the mapping of VHDL to B and Communicating Sequential Processes (CSP) methods. The author proposes to derive a B model from VHDL and formalize requirements with CSP. This approach uses a model-checking technique that requires modification and re-checking of the implementation until the desired integrity level is achieved.

A BHDL tool has been proposed for digital circuit design [64]. The tool converts a VHDL description into a B specification with two machines: an abstract that represents a VHDL entity and an implementation that corresponds to the architecture. Then, these two machines are verified using the B engine and the VHDL comments are interpreted as invariant properties. In contrast to this approach, we derive an implementable deterministic Event-B model following the usual refinement-based development. Then, components are injected into the model, so that a structural VHDL description can be generated.

Instead of concentrating on the derivation of formal models from implementations, we focus on the code generation from the formal models. We employ the Event-B design methodology, where the model development follows the refinement approach and eventually ends in code generation in an automated manner. In addition to behavioural VHDL description generation, we propose to translate a model into a description with components (i.e., a structural description) in order to derive a more efficient implementation.

6 Discussion and Research Directions

In this chapter, we conclude the work described in the thesis. Additionally, we discuss the limits of the proposed approaches and outline future research directions.

6.1 Conclusion

As parallel and distributed computing becomes central in modern computations, many-core platforms are envisaged to be used in various application domains including critical ones. To provide resilience of the platform and maintain its performance at an acceptable level, we have designed an agent-based management system that monitors the state of the platform and applies various dynamic reconfiguration mechanisms when necessary. Considering hierarchical formation of the agents, the (re)configuration procedures have been evenly distributed such that effective and efficient monitoring and recovery are possible. We have developed algorithms for initial configuration (mapping) and dynamic reconfiguration (tasks reallocation). The initial mapping algorithm provides each application with a number of spare resources whilst the tasks reallocation algorithm utilizes these spare resources in an intelligent manner.

The proposed agent-based management system with (re)configuration mechanisms has been developed following the formal refinement approach, so that its behaviour can justifiably be trusted. In other words, the use of formal specification and verification methods help us to mathematically ensure that the developed system behaves correctly with respect to the specified properties. Moreover, we have presented the guidelines that help designers to build such complex systems in a rigorous manner.

When tasks are dynamically reallocated, they may lose data which may lead to the production of incorrect result. To tackle this problem, we have proposed a scalable mechanism in which the communication between tasks uses duplicate packets. We have presented algorithms for the intelligent packet handling considering different types of tasks.

A natural logical step after the specification of a system is derived lies in an automated code generation. This lowers the probability of introducing design faults and facilitates easier evaluation of non-functional requirements in a real-world environment. To address this problem, we have proposed mechanisms that allow designers to generate a behavioural (without components) or a structural (with components) hardware description directly from a formal model.

Since efficiency is one of our objectives, we have also evaluated performance of the proposed mechanisms on a commercially available platform TilePro by Tilera [3]. The evaluation results have illustrated that the proposed algorithms produce a marginal overhead and perform efficiently while allowing applications to produce the expected result.

Therefore, the main goal of the thesis has been accomplished by proposing a design flow for the development of complex systems such as agent-based management systems. This design flow starts with a formal rigorous modelling of a system and ends in an automated code generation.

6.2 Future work

Although the evaluation results show efficiency of the proposed system in terms of performance and area overhead, the number and the placement of spare cores affects the utilization and the performance of the underlying platform. Thus, one research direction includes exploration of possible placements of spare resources within an application region in order to find an optimal solution. For instance, spare resources can be placed randomly following the approach presented in [39]. In addition, we will investigate reallocation of tasks to unallocated spare cores within other clusters.

The proposed dynamic reconfiguration procedures constitute one part of resilience to failures. The other part requires techniques to detect these failures. Hence, another direction of our research is to explore failure detection mechanisms. In particular, various techniques such as model-based diagnosis [65] or runtime verification [66] can be integrated into the agents in order to provide comprehensive and fast failure detection. This will allow for rapid invocation of the proposed dynamic reconfiguration procedures.

As mentioned above, the rigorous formal development provides means to insure correct behavior of a system with respect to postulated properties by utilizing mathematical proofs. Moreover, the automated code generation prevents a designer from the introduction of design faults into the system while implementing/coding the derived specification. However, the derived implementation is run on processing units as well. Despite the fact that the agents are simpler than the application tasks, physical failures of PUs can occur as well. Hence, resilience of the agent-based management system can also be improved by considering these failures and dynamic reconfiguration of the agents.

Finally, the system composition of software and hardware poses the question of SW/HW co-design, where implementations of both parts can be obtained in an automated manner. Hence, formal rigorous SW/HW co-design has also a particular research interest. This affects the system design flow in the following directions. The first one is the decomposition of Event-B models which gives us an opportunity to construct a hierarchical structure of a model, so that this structure is reflected in the implementation. The second direction is the introduction of combinatorial components that depend on the clock signal and allow a designer to derive a time-aware model as well as to generate synchronous code from this model.

Bibliography

[1] K. Asanovic, R. Bodik J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A View of the Parallel Computing Landscape, Communications of the ACM, Vol. 52, No. 10, pp. 56-67, 2009.

[2] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, Computer, IEEE, Vol. 35, Issue 1, pp. 70 – 78, 2002.

[3] Tilera, Tile Processor Architecture Overview, 2011. Available: http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf

[4] Intel, Single-Chip Cloud Computer: Project, 2013. Available: http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-program-guide.pdf

[5] F. Khalili, H. R. Zarandi, A Fault-Tolerant Low-Energy Multi-Application Mapping onto NoC-based Multiprocessors, International Conference on Computational Science and Engineering, Nicosia, IEEE, pp. 421-428, 2012.

[6] G. Link, N. Vijaykrishnan, Hotspot Prevention Through Runtime Reconfiguration in Networks-on-Chip, DATE, IEEE, pp. 648-649, 2005.

[7] S. Mukherjee, J. Emer, S. Reinhardt, The Soft Error Problem: An Architectural Perspective, International Symposium on High-Performance Computer Architecture, IEEE, pp. 243-247, 2005.

[8] C. Bolchini, M. Carminati, A. Miele, Self-Adaptive Fault-Tolerance in Multi-/Many-Core Systems, Journal of Electronic Testing: Theory and Applications, Vol. 29, Issue 2, Springer US, pp. 159-175, 2013.

[9] K. Motamedi, N. Ionnides, M. Rümmeli, I. Schagaev, Reconfigurable Network on Chip Architecture for Aerospace Applications, Preprints of the 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software, pp. 131-136, 2009.

[10] P. Rantala, J. Isoaho, H. Tenhunen, Novel Agent-Based Management for Fault-Tolerance in Network-on-Chip. Euromicro Conference on Digital System Design Architectures, Methods and Tools, Lubeck: IEEE pp. 551-555, 2007.

[11] A. Yin, L. Guang, P. Liljeberg, E. Nigussie, J. Isoaho, H. Tenhunen, Hierarchical Agent Based NoC with Dynamic Online Services, Industrial Electronics and Applications, Taichung: IEEE, pp. 434-439, 2009.

[12] L. Guang, Hierarchical Agent-based Adaptation for Self-Aware Embedded Computing Systems, PhD Thesis, University of Turku, 2012.

[13] Y. Kawabe, K. Mano, K. Kogure, The Nepi^2 Programming System: A π-Calculus-based Approach to Agent-based programming, International Workshop on Formal Approaches to Agent-based systems, pp. 90-102, 2001.

[14] R. J. Back and J. Wright, Refinement Calculus: A Systematic Introduction,

Springer-Verlag, 1998.

[15] J.-R. Abrial, Modeling in Event-B: System and Software Engineering. Cambridge: Cambridge University Press, 2010.

[16] RODIN, 2014. Available: http://sourceforge.net/projects/rodin-b-sharp/

[17] S. Wright, Automatic Generation of C from Event-B, Workshop on Integration of Model-based Formal Methods and Tools, p. 14, 2009.

[18] A. Edmunds, M Butler, Linking Event-B and Concurrent Object-Oriented Programs, Electronic Notes in Theoretical Computer Science 214, pp. 159-182, 2008.

[19] D. Méry, N. K. Singh, Automatic Code Generation from Event-B models, Symposium on Information and Communication Technology, ACM, pp. 179-188, 2011.

[20] M. Dehyadgari, M. Nickray, A. Afzali-Kusha, Z. Navabi, Evaluation of pseudo adaptive XY routing using an object oriented model for NoC, International Conference on Microelectronics, IEEE, pp. 204-208, 2005.

[21] V. Rantala, T. Lehtonen, J. Plosila, Network on Chip Routing Algorithms, TUCS Technical Report 779, pp. 10-16, 2006.

[22] M. Ebrahimi, D. Masoud, P. Liljeberg, J. Plosila, H. Tenhunen, Efficient Congestion-Aware Selection Method for On-Chip Networks, International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), IEEE, pp. 1-4, 2011.

[23] Tilera, Tile Processor User Architecture Manual, 2011. Available: http://www.tilera.com/scm/docs/UG101-User-Architecture-Reference.pdf

[24] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H.Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, S. Borkar, An 80-tile sub-100-w teraflops processor in 65-nm cmos, IEEE JSSC, 43(1), pp. 29–41, 2008.

[25] I. Khatib, D. Bertozzi, F. Poletti, L. Benini, A. Jantsch, M. Bechara, H. Khalifeh, M. Hajjar, R. Nabiev, S. Jonsson, MPSoC ECG biochip: a multiprocessor system-on-chip for real-time human heart monitoring and analysis, Conference on Computing Frontiers, New York: ACM, pp. 21-28, 2006.

[26] J.-C. Laprie, From Dependability to Resilience, International Conference on Dependable Systems and Networks, IEEE/IFIP, pp. G8-G9, 2008.

[27] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, pp. 11-33, 2004.

[28] A. Avizienis, J.-C. Laprie, B. Randell, Fundamental Concepts of Dependability, 2001. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.6074

[29] S. Russel, P. Norvig, Artificial Intelligence: A Modern Approach (2nd edition), Prentice Hall, Englewood, p. 946, 2003.

[30] C. Métayer, J.-R. Abrial, L. Voisin, Deliverables, Rigorous Open Development Environment for Complex Systems, 2005. Available: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf

[31] K. Robinson, System Modelling & Designing using Event-B, 2010. Available: http://wiki.event-b.org/images/SM%26D-KAR.pdf

[32] Simulink, Simulation and Model-Based Design, 2014. Available: http://www.mathworks.se/help/simulink/index.html

[33] MathWorks, Modeling Dynamic Systems, 2014. Available: http://www.mathworks.se/help/simulink/ug/modeling-dynamic-systems.html

[34] MathWorks, Simulating Dynamic Systems, 2014. Available: http://www.mathworks.se/help/simulink/ug/simulating-dynamic-systems.html

[35] MathWorks, Simulink coder, 2013. Available: http://www.mathworks.se/products/simulink-coder/

[36] IEEE Standard VHDL Language Reference Manual, IEEE 1076, 2008.

[37] Altera, Quartus-II software, 2014. Available: http://www.altera.com/products/software/sfw-index.jsp

[38] C. Pinello, L. Carloni, A. Sangiovanni-Vincentelli, Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications, International Conference on Design Automation and Test in Europe, IEEE, pp. 1164-1169, 2004.

[39] C.-L. Chou, R. Marculescu, FARM: Fault-Aware Resource Management in NoC-based Multiprocessor Platforms, DATE Conference & Exhibition, Grenoble, IEEE, pp. 1-6, 2011.

[40] Xilinx, Remote FPGA Reconfiguration Using MicroBlaze or PowerPC Processors, 2006. Available:
http://www.xilinx.com/support/documentation/application_notes/xapp441.pdf

[41] Altera, Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs, 2010. Available:
http://www.altera.com/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf

[42] J. Ceponis, E. Kazanavicius, A. Mikuckas, Fault Tolerant Process Networks, Information Technology and Control, Vol. 35, No. 2, pp. 124-130, 2006.

[43] P. Hölzenspies, T. Braak, J. Kuper, G. Smit, J. Hurink, Run-time Spatial Mapping of Streaming Applications to Heterogenous Multi-Processor Systems, International Journal on Parallel Programming, pp. 68-83, 2009.

[44] S. Le Beux, G. Bois, G. Nicolescu, Y. Bouchebaba, M. Langevin, P. Paulin, Combining mapping and partitioning exploration for NoC-based embedded systems, Journal of Systems Architecture, New York: Elsevier, pp. 223-232, 2010.

[45] C. Dima, A. Girault, C. Lavarenne, Y. Sorel, Off-line real-time Fault-Tolerant Scheduling, Euromicro, IEEE, pp. 410-417, 2001.

[46] C. Andres, C. Molinero, M. Nuez, A formal methodology to specify

hierarchical agent-based systems, Signal Image Technology and Internet Based Systems, Bali: IEEE, pp. 169-176, 2008.

[47] G. Ali, N. Zafar, Modelling Agent-Based Systems Using X-Machine and Z Notation, International Communication Software and Networks, Singapore: IEEE, pp. 249-253, 2010.

[48] A. Lanoix, Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles, International Symposium on Theoretical Aspects of Software Engineering, Nanjing: IEEE, pp. 297-304, 2008.

[49] T. Araragi, P. Attie, I. Keidar, K. Kogure, V. Luchangco, N. Lynch, K. Mano, On Formal Modeling of Agent Computations, Lecture Notes in Computer Science, Berlin: Springer-Verlag, pp. 48-62, 2001.

[50] T. Araragi, Agent programming and its formal specification, Technical Report ai99-47, pp. 47-54, 1999.

[51] N. Lynch, M. Tuttle, An introduction to I/O automata, CWI-Quarterly 2(3), pp. 219-246, 1989.

[52] T. Seceleanu, Systematic Design of Synchronous Digital Circuits, Turku: TUCS Dissertations, Turku Centre for Computer Science, 2001.

[53] S. Hallerstede, Y. Zimmermann, “Circuit Design by Refinement in Event-B”, FDL, pp. 624-637, 2004.

[54] M. Benveniste, A «Correct by Construction» Realistic Digital Circuit, RIAB Workshop, FMWeek, 2009.

[55] I A. Benveniste, P. Le Guernic, Hybrid Dynamical Systems Theory and the Signal Language, IEEE Transactions on Automatic Control 35(5), pp. 535-546, 1990.

[56] D. Potop-Butucaru, R. de Simone, Optimizations for Faster Execution of Esterel Programs, Proc. of MEMOCODE conference, pp. 227-236, 2003.

[57] I. Sander, A. Jantsch, System Modelling and Transformational Design Refinement in ForSyDe, Transactions on Computer Aided Design of Integrated Circuits and Systems, IEEE, Vol. 23, 2004, pp. 17-32.

[58] J. Talpin, P. Guernic, S. Shukla, R. Gupta, F. Doucet, Polychrony for Formal Refinement Checking in a System-Level Design Methodology, Application of Concurrency to System Design (ACSD), IEEE, pp. 9-19, 2003.

[59] University of California, BlueSpec Documentation, 2008. Available: http://www.ece.ucsb.edu/its/bluespec/index.html.

[60] D. Richards, D. Lester, A monadic approach to automated reasoning for BlueSpec SystemVerilog, Innovations System Software Engineering, Springer-Verlag, pp. 85-95, 2011.

[61] G. Singh, E. Shukla, Verifying Compiler-based Refinement of Bluespec Specifications using the SPIN model Checker, 15th International SPIN Workshop, Springer, pp. 250-269, 2008.

[62] R. J. Back, K. Sere, Superposition Refinement of Reactive Systems, Formal Aspects of Computing, Springer, Vol. 8, 1995, pp. 324-346.

[63] N. Evans, Integrating Formal Methods with Informal Digital Hardware Development, Proc. of AVoCS, p. 1-16, 2010.

[64] A. Aljer, P. Devienne, S. Tison, BHDL: Circuit design in B, Conference on Application of Concurrency to System Design, IEEE, pp. 1-2, 2003.

[65] R. Isermann, Model-based fault-detection and diagnosis – status and applications, Annual Reviews in Control 29(1), Elsevier, Vol. 29, Issue 1, pp. 71-85, 2005.

[66] L. Pike, S. Niller, N. Wegmann, Runtime Verification for Ultra-Critical Systems, In Proceedings of International Conference on Runtime Verification, Springer, pp. 310-324, 2012.

Part II

Research Publications

Paper 1

Hierarchical agent-based monitoring systems for dynamic reconfiguration in NoC platforms: A formal approach

Sergey Ostroumov, Leonidas Tsiopoulos, Marina Waldén, Juha Plosila

Originally published in:
Advancing Embedded Systems and Real-Time Communications with Emerging Technologies, Chapter 13, IGI Global, pp. 302-333, 2014.

Based on the publication:
Sergey Ostroumov, Leonidas Tsiopoulos, Formal Development of Hierarchical Agent-Based Monitoring Systems for Dynamically Reconfigurable NoC Platforms, International Journal of Embedded and Real-Time Communication Systems (IJERTCS), Volume 3, Issue 2, pp. 40–72, 2012.

© The chapter appears in Advancing Embedded Systems and Real-Time Communications with Emerging Technologies edited by Seppo Virtanen. Copyright 2014, IGI Global, www.igi-global.com. Posted by permission of the publisher.

Paper 2

Formal Approach to Agent-based Dynamic Reconfiguration in Networks-on-Chip

Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere

Originally published in:
Journal of Systems Architecture, Volume 59, Issue 9, Elsevier, pp. 709-728, 2013.

© Reprinted from Embedded Software Design Journal of Systems Architecture, 59(9), Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Formal Approach to Agent-based Dynamic Reconfiguration in Networks-on-Chip, p. 20, 2013, with permission from Elsevier.

Journal of Systems Architecture 59 (2013) 709–728

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Formal approach to agent-based dynamic reconfiguration in Networks-On-Chip

Sergey Ostroumov [a,b,*], Leonidas Tsiopoulos [b], Juha Plosila [c], Kaisa Sere [b]

[a] TUCS – Turku Centre for Computer Science, Turku, Finland
[b] Åbo Akademi University, Department of IT, Joukahaisenkatu 3-5a, Turku 20520, Finland
[c] University of Turku, Department of IT, Joukahaisenkatu 3-5b, Turku 20014, Finland

ARTICLE INFO

Article history:
Available online 17 June 2013

Keywords:
Agent-based system
Dynamic reconfiguration
Event-B
Formal methods
Fault-tolerance
Network-On-Chip

ABSTRACT

A Network-On-Chip (NoC) platform is an emerging topology for large-scale applications. It provides a required number of resources for critical and excessive computations. However, the computations may be interrupted by faults occurring at run-time. Hence, reliability of computations as well as efficient resource management at run-time are crucial for such many-core NoC systems. To achieve this, we utilize an agent-based management system where agents are organized in a three-level hierarchy. We propose to incorporate reallocation and reconfiguration procedures into agents hierarchy such that fault-tolerance mechanisms can be executed at run-time. Task reallocation enables local reconfiguration of a core allowing it to be eventually reused in order to restore the original performance of communication and computations. The contributions of this paper are: (i) an algorithm for initial application mapping with spare cores, (ii) a multi-objective algorithm for efficient utilization of spare cores at run-time in order to enhance fault-tolerance while maintaining efficiency of communication and computations at an adequate level, (iii) an algorithm integrating the local reconfiguration procedure and (iv) formal modeling and verification of the dynamic agent-based NoC management architecture incorporating these algorithms within the Event-B framework.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Excessive many-core computations require a large number of resources to be available at their disposal. Critical applications, in their turn, require fault-tolerance mechanisms that can be executed at run-time so that the computations can continue without interruption. In addition, the computations have to be performed in an efficient manner. Hence, it is necessary to provide a platform and means that would satisfy these requirements.

A Network-On-Chip (NoC) platform is an emerging topology for large-scale applications [1]. It provides a desired number of resources for critical and excessive computations from, for example, biomedical [2] or aerospace domain [3]. However, special means are required to monitor the state of the platform and to apply dynamic procedures for tolerating faults. These means are usually implemented in the form of agents [4]. The agents help to avoid overloading the NoC platform with monitoring and recovering activities while the platform performs routing algorithms, etc. The bigger the platform, the more agents it requires. In order for the system to manage a large number of agents, these agents are organized in a hierarchy, typically of a multi-level structure [5]. This hierarchy usually consists of the platform (system) agent managing the whole platform, cluster agents operating on certain regions (i.e., sets of cores where applications are mapped) and cell agents processing local (cell) information. The hierarchy permits the agents to exchange the data about the current state of the platform as well as to tolerate faults by applying run-time reconfiguration procedures at different levels.

The use of NoC platforms with their run-time management systems in critical applications requires these platforms to be reliable. One of the appropriate approaches for specifying and verifying reliable NoC systems is provided by formal methods. Formal development enables stepwise and correct-by-construction design of the specification of a system allowing mathematical reasoning of its correctness. Moreover, formal methods are recommended by safety standards [6] for the development of safety critical systems which we envisage to be one of the application domains for the system we develop in this paper. We adopt the Event-B formalism [7] as the primary framework for the formal development. Event-B supports system level modeling as well as supplies a proving mechanism to reason about the correctness of the specification w.r.t. the functional properties (requirements) postulated as invariants. The specification within Event-B is created following

[*] Corresponding author at: Åbo Akademi University, Department of IT, Joukahaisenkatu 3-5a, Turku 20520, Finland. Tel.: +358 215 3339.
E-mail address: Sergey.Ostroumov@abo.fi (S. Ostroumov).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.06.001

the refinement-based approach, i.e., incremental unfolding of system properties supported by mathematical proofs. Furthermore, it has adequate tool support through the Rodin platform [8].

In our approach, we propose to incorporate dynamic reallocation and reconfiguration procedures at different levels of the agents hierarchy. In particular, we show a specific initial application mapping to a region with spare cores at the platform agent level. The platform agent also assigns a cluster agent to each application. Additionally, the platform agent can remap the entire application or reallocate a task of a particular cell outside the application region if all the spare cores within the cluster have been utilized and there is a new fault. However, the platform may contain a large number (thousand) of cores with many applications mapped on such a platform. Hence, the reallocation of a task has to be performed efficiently while still allowing efficient execution of an application. To achieve this, a corresponding cluster agent utilizes the available spare cores when moving a task from a faulty core to a spare one within the application region. This allows a more efficient task migration in terms of reallocation speed and power consumption than task migration to free cores allocated at a great distance from the region. After the task migration is complete, a local cell agent initiates the local reconfiguration procedure that enables the faulty core to recover its functionality and to be reinvolved in the computations. This permits the region and, hence, the application to restore the original performance of the computations. The specification of this system is developed within Event-B and supported by mathematical proofs of its correctness.

The contributions of this paper are (i) an algorithm for the initial application mapping and tasks allocation with free spare cores, (ii) a multi-objective algorithm that facilitates fault-tolerance of the platform while maintaining performance of communication and computations at an adequate level, (iii) an algorithm integrating the local reconfiguration procedure and (iv) the Event-B formal modeling and verification of a hierarchical agent-based dynamic management architecture for NoCs incorporating these algorithms. In this architecture, the platform agent dynamically creates and destroys the cluster agents. The cluster agents, in their turn, are fully distributed, i.e., they independently of each other execute monitoring and reconfiguration procedures utilizing the spare cores without overloading the platform agent. We follow the refinement approach where the base system model has been derived from a previously developed model [9].

The remainder of the paper is organized as follows. In Section 2, we review the related work. In Section 3, we overview the Event-B formal framework, briefly describe agent-based management systems for NoC platforms and present an approach to formal development of such systems. In Section 4, we give the algorithms for the reallocation and reconfiguration procedures performed at different levels of the hierarchy. In Sections 5, 6 and 7 we formally develop the specifications of the platform, cluster and cell agents through refinement, respectively. Finally, we conclude the paper and highlight the directions of the future work in Section 8.

2. Related work

A fault-tolerant reconfigurable NoC has been proposed by Motamedi et al. [3]. The authors consider application specific architecture for avionic systems. In particular, they use a star network topology as the main active formation where the so called cockpit switch is placed in the center of the topology. The redundancy is achieved by placing redundant links in the system. When a fault is detected, the topology is switched (reconfigured) from the star formation to the ring one. In addition, the authors utilize an Embedded Reliable Reduced Instruction Processor (ERRIC) as a computational unit. ERRIC has been specially designed for permanent faults. Additionally, the authors show the prototyping results where the overhead of using their approach is marginal while the required level of fault-tolerance is achieved. Although ERRIC is used as a computational unit, it has a reduced instruction set which may not be applicable to application domains other than avionics. Moreover, ERRIC is implemented on Field-Programmable-Gate-Array (FPGA) or Application-Specific-Integrated-Circuit (ASIC), where a fault may also occur and, hence, this processing unit is not available any more.

In comparison, instead of reconfiguring the NoC topology, we consider a topologically fixed NoC platform that is not application specific. We note however that the approach we propose in this paper can be applied to any topology and any type of routing schemes since it does not depend on the underlying platform. Our approach allows for executing applications without interruption and recovering the functionality of the platform by applying dynamic task reallocation and local cell reconfiguration procedures, respectively. The local reconfiguration, which is executed on the processing unit instead of the topology, recovers the operational mode of the former. Nevertheless, redundant routers (and/or links) can complement our approach.

A three-level architecture for agent-based monitoring of the NoC platform has been proposed by Guang et al. [10]. The approach allows for effective monitoring of the state of the NoC platform. The authors present a structured framework for designing such a system. However, the framework only describes the main definitions of the hierarchical agent-based system and do not consider faults of the NoC elements. Moreover, the framework is not formally verified.

Guang et al. [11] have also proposed to incorporate reconfiguration procedures at coarse-grained (system) and fine-grained (local) levels for tolerating permanent and transient faults in many-core (thousand-core) Systems-On-Chip. They have suggested a two-level architecture where the system agent manages the whole platform and the local agent monitors the local component such as a router. The system also uses a portion of spare cores that are utilized if some processor fails. However, the authors do not show where these spare cores are located and do not describe the algorithm of utilization of these spare cores. This may lead to a problem of drastically decreased performance, if these spares are at a great distance from applications running computations. Moreover, the authors only consider the faults of the routers in which case the local agent executes reconfiguration by replacing a broken wire with a spare one. In addition, the functionality of the system agent includes many activities that may lead to a failure state of the agent itself, although the system agent is designed with higher reliability.

In contrast to [10,11], we adopt a three-level architecture where reconfiguration procedures are incorporated into different levels of the hierarchy such that the platform can be dynamically adapted and healed, if necessary. In particular, the platform agent can remap the entire application or reallocate a task of a particular cell within the platform. This only occurs if all the spare cores within the cluster have been utilized and there is a new fault. Since the platform may contain a large number of cores (thousand-core) and many applications can be mapped on such a platform, the reallocation of a task has to be performed efficiently while still allowing efficient execution of an application. Hence, we propose to introduce the reallocation procedure within the region into the functionality of a cluster agent, which is responsible for its cluster (region). These agents are dynamically created and destroyed when an application is mapped to and released from the platform, respectively. Finally, the local cell agent executes the local reconfiguration procedure allowing the cell to recover. In our opinion, the three-level architecture we propose provides better scalability and structure for complex NoC platforms. Moreover, the functionality of the agents in this architecture is more balanced enhancing dependability of the platform and not overloading the platform (system) agent. Furthermore, our approach has been developed following the refinement-based and correct-by-construction approach allowing formal verification by discharging proof obligations.

A similar approach to remapping with spare cores has been proposed by Chou and Marculescu [12]. The authors study three possible schemes of spare cores allocation at the system level only, without considering assignments of spare cores within application/cluster regions. The three possible assignments include: (1) side assignment, (2) random assignment and (3) uniform assignment. The authors provide the metrics for evaluation of the task remapping to spare cores and point out that the remapping to the randomly placed spare cores performs better than to the spare cores placed to the side of the system. Clearly, a spare core allocated at a great distance from an application drastically decreases the entire system performance.

In contrast, we propose to incorporate spare cores at the side of each cluster (region) instead of spare cores assignment at the system level only. We note however that random (or other types of) assignment of spare cores within the region is out of the scope of this paper. Depending on the size of an application, a fixed number of spare cores is provided to a corresponding cluster agent allowing it to tolerate faults while maintaining the performance of the computations at an adequate level. We provide an algorithm for spare cores utilization at the cluster level. In addition, we propose to initiate a local reconfiguration procedure on a faulty cell in order to recover its functionality. When this procedure is complete, the cluster agent reallocates the task back restoring the original performance of computations.

There are several other works proposing dynamic (re)mapping of applications. Some of them are single-objective, i.e., they focus on minimizing, for instance, energy consumption [13]. Other works address simultaneous optimization of mapping and software-hardware partitioning without considering faults of the platform [14]. In our approach, we propose to integrate and uniformly distribute the reallocation and reconfiguration functionality within the agents hierarchy such that a higher level of fault-tolerance is achieved while performance remains at an adequate level. Furthermore, to the best of our knowledge, all of these approaches have been developed informally, w.r.t. correct-by-construction and proof-based development, while our approach is supported by applying the Event-B formal framework through refinements and correctness proofs.

3. Preliminaries and proposed approach

3.1. The Event-B formalism

Event-B is a formalism for stepwise and correct-by-construction development of a system [7]. A specification in Event-B consists of two parts: a context and a machine. The context can be extended by another context while the machine can be refined by another machine. In addition, the machine can refer to the context data, if this machine sees this context.

The context defines the static part of the model – data types (sets), constants, and their properties given as a collection of axioms. The machine describes the dynamic behavior of the system in terms of its state (model or state variables) and state transitions, called events. The essential and guaranteed system properties are formulated as invariants.

The machine is uniquely identified by its name <**machine identifier**>. The state variables of the machine are declared in the **variables** clause and initialized in the **initialisation** event. The variables are strongly typed by constraining predicates given in the **invariants** clause. The overall system invariant is defined as a conjunction of constraining predicates and the other predicates stating the functional system properties that should be preserved during system execution. The machine may contain so-called **convergent** events that are executed several times in a row. These events must eventually terminate in order for other (non-convergent) events to take place. This fact is assured by a variant introduced into the **variant** clause. The variant represents a natural number (or a finite set) whose value (or cardinality) is decreasing each time a convergent event is executed. The behavior of the system is then defined by a collection of atomic events specified in the **events** clause. The syntax of an event is as follows:

$$\mathsf{E} = \textbf{ANY}\ \mathsf{x}\ \textbf{WHERE}\ \mathsf{g}\ \textbf{THEN}\ \mathsf{a}\ \textbf{END}$$

where x is a list of event local variables, the guard g is a conjunction of predicates over the state variables and the local variables and the action a is a collection of assignments to the state variables.

The guard is a predicate that determines the conditions under which the action can be executed, i.e., when the event is enabled. If several events are enabled simultaneously, then any of them can be chosen for execution non-deterministically. If none of the events is enabled, then the system deadlocks.

The action of an event is a composition of assignments executed simultaneously and denoted as ||. An assignment to a variable can be either deterministic or non-deterministic. A deterministic assignment is defined as $\mathsf{y} := \mathsf{E(v)}$, where y is a list of the state variables and $\mathsf{E(v)}$ is a list of expressions over the state variables v. A non-deterministic assignment is specified as $\mathsf{y} :| \mathsf{Q(v, y')}$, where $\mathsf{Q(v, y')}$ is a predicate and the primed variable $\mathsf{y'}$ represents a new value the variable y gets after the event execution. As the result of a non-deterministic assignment, the variable y gets such a value $\mathsf{y'}$ that $\mathsf{Q(v, y')}$ holds.

The semantics of Event-B events is defined using before-after (BA) predicates [15]. An action of an event is seen as a BA that describes a relationship between the system state before (v) and after ($\mathsf{v'}$) the execution of the event. The formal semantics provides us with a foundation for establishing system correctness. To verify correctness (consistency) of a specification, we should discharge a number of proof obligations. In particular, each event of the model should be consistent with the invariant preservation proof obligation (INV) whereas every event that contains a non-deterministic assignment should also satisfy event feasibility (FIS):

$$\mathsf{Inv} \wedge \mathsf{g_e} \Rightarrow [\mathsf{BA_e}]\mathsf{Inv} \qquad \text{(INV)}$$

$$\mathsf{Inv} \wedge \mathsf{g_e} \Rightarrow \exists \mathsf{v'} \cdot \mathsf{BA_e} \qquad \text{(FIS)}$$

where Inv is a model invariant, $\mathsf{g_e}$ and $\mathsf{BA_e}$ are the guard and the before-after predicate of the event e, respectively, and $[\mathsf{BA_e}]\mathsf{Inv}$ stands for the substitution in the invariant Inv according to the before-after predicate $\mathsf{BA_e}$ of the event e.

When modeling a continuous procedure (e.g., a loop), some events may be executed several times in a row (convergent events). To guarantee that the number of times when such convergent events are executed is finite, one has to provide a variant and show the consistentency of these events with the following proof obligation [16]:

$$\forall \mathsf{S}, \mathsf{C}, \mathsf{V} \cdot \mathsf{A} \wedge \mathsf{I} \Rightarrow \mathsf{finite(Var)} \wedge \mathsf{card}([\mathsf{BA_e}]\mathsf{Var}) < \mathsf{card(Var)} \qquad \text{(VAR)}$$

where S and C represent sets and constants introduced into contexts, respectively. V stands for a set of state variables. A is a collection of axioms. I depicts a set of invariants. Var is an expression (a variant) that denotes a finite set of values.

The specification within Event-B is developed in a stepwise manner through refinements. Invariance properties are preserved by refinement, hence, do not require to be re-proved. However, concrete (refined) events must be able to simulate their abstract counterparts according to some gluing invariant. This is formally ensured by discharging the guard strengthening (GRD) and action simulation (SIM) proof obligations [16]:

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \wedge g_r \Rightarrow g \qquad \text{(GRD)}$$

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \cdot A \wedge A_r \wedge I \wedge I_r \wedge BA_{er} \Rightarrow BA_e \qquad \text{(SIM)}$$

where all letters with the subscript “r” stand for the refined versions of the structures described above.

The Rodin platform [8], a tool supporting Event-B, automatically generates the required proof obligations and attempts to automatically discharge (prove) them. Sometimes it requires user assistance that is provided via the interactive prover. However, in general the tool achieves high level of automation (usually over 80%) in proving.

3.2. Agent-based monitoring systems for NoCs

NoC is generally considered as an efficient and scalable interconnect paradigm [17]. It allows sophisticated applications to be deployed on many-core platforms and execute their intensive computations effectively. The cores in an NoC platform are interconnected with one structured net that permits the cores to achieve a high level of communication performance. Hence, these systems are likely to be used in many applications, especially critical ones.

Critical applications [2,3] require their computations to continue without interruption even when a fault occurs. Depending on the size and the purpose of an application, it may take a large amount of time for its computations [18], which increases the probability of faults. Moreover, fault occurrence is rising on many-core systems because of increasing resource integration [19]. Due to the criticality of such applications, an NoC platform has to provide necessary resources as well as redundancy. Hence, the platform has to implement special means facilitating efficiency, redundancy and dynamic reconfiguration. These means are represented by agents that monitor the state of the platform and apply necessary mechanisms statically and dynamically.

The number of these agents grows with the size of the platform. To manage a large number of agents, they are organized in a hierarchy that typically has a three-level structure generally applied to a 2D mesh topology [4,5,10]. An example of the architecture we propose for the system is shown in Fig. 1. In this architecture, a cell has a heterogeneous structure that contains a control device, i.e., a local cell agent, and a reconfigurable core: fine-grained (see for example [20]) or coarse-grained reconfigurable regions (see for example [21]). It may also contain other dedicated hardware blocks for faster execution. This structure facilitates dynamic reconfiguration and recovery of a cell. However, all the cells in the platform have the same heterogeneous structure. This allows for efficient mapping and task migration. A local cell agent monitors the state of a corresponding cell and can change the cell behavior, if necessary. Cluster agents are dynamically created when an application is mapped onto the platform. They manage regions where applications are mapped by monitoring and, e.g., adjusting regional parameters such as frequency and/or voltage. The platform agent, which is persistent in the system, manages the whole platform. It creates and destroys cluster agents while mapping and releasing applications to and from the platform, respectively.

3.3. Base formal model of agent-based management system

Let us now briefly describe what we have done in our previous work relating to such agent-based management architectures. In [9], we have developed a three-level hierarchy of the agents through refinements. Fig. 2 illustrates the hierarchy of the agents and their communication with shared variables. This system serves as the base of the system we develop in this paper.

The developed model proposed in [9] is considered generic such that it can be instantiated and further developed in accordance to specific requirements. More specifically, applications have been

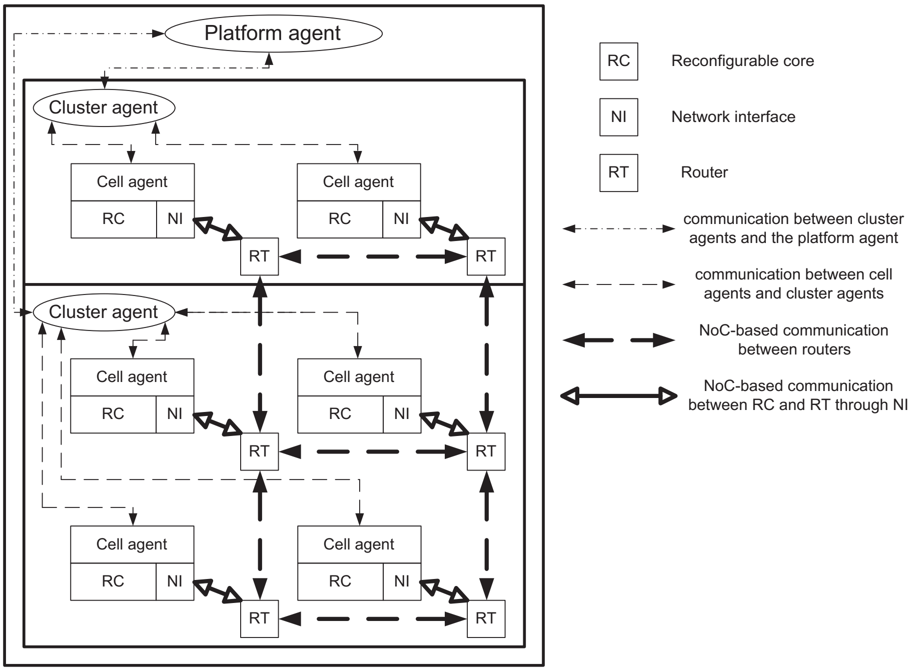

Platform agent
Cluster agent
Cell agent
RC
NI
RT
Cell agent
RC
NI
RT
Cluster agent
Cell agent
RC
NI
RT
Cell agent
RC
NI
RT
Cell agent
RC
NI
RT
Cell agent
RC
NI
RT
RC
Reconfigurable core
NI
Network interface
RT
Router
communication between cluster agents and the platform agent
communication between cell agents and cluster agents
NoC-based communication between routers
NoC-based communication between RC and RT through NI

Fig. 1. NoC architecture with agents.

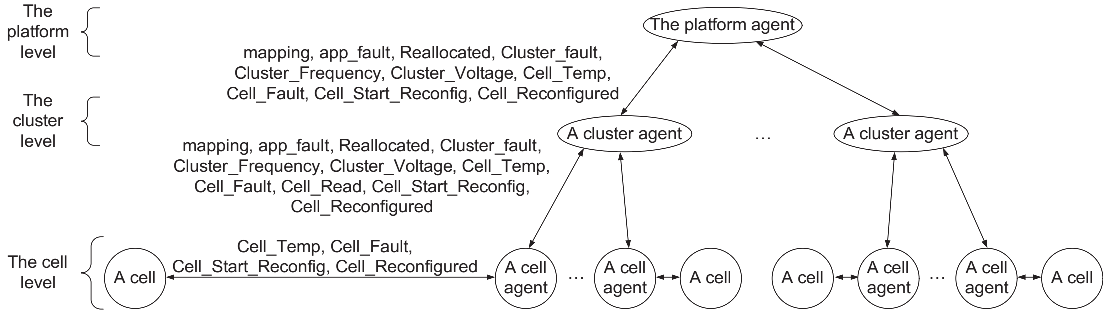

The platform level
The cluster level
The cell level
The platform agent
mapping, app_fault, Reallocated, Cluster_fault, Cluster_Frequency, Cluster_Voltage, Cell_Temp, Cell_Fault, Cell_Start_Reconfig, Cell_Reconfigured
A cluster agent
...
A cluster agent
mapping, app_fault, Reallocated, Cluster_fault, Cluster_Frequency, Cluster_Voltage, Cell_Temp, Cell_Fault, Cell_Read, Cell_Start_Reconfig, Cell_Reconfigured
Cell_Temp, Cell_Fault, Cell_Start_Reconfig, Cell_Reconfigured
A cell
A cell agent
...
A cell agent
A cell
A cell
A cell agent
...
A cell agent
A cell

Fig. 2. The three-level hierarchy of the agents.

modeled as a deferred set, i.e., they were given to the platform as an abstract data type without considering application task graphs [18,22] for efficient mapping. Furthermore, the mapping itself is modeled as a simplistic function that mapped applications onto rectangular shaped regions without spare cores whose area is computed as $\mathsf{area} = \mathsf{x} * 2$, such that $\mathsf{x} = \lceil \mathsf{n}/2 \rceil$, where x is a number of columns and n is the number of requested resources. This function is mainly to show the implementation of the simple procedures for resource searching. In addition, the platform agent can remap the whole application to another region, if the whole region where an application is mapped is faulty and there is another region where the application can be remapped. If there exists no such a free region, the platform agent can perform a task migration procedure from a faulty cell to another cell within the platform nondeterministically. This procedure may not be efficient for large-scale platforms due to the fact that the reallocation procedure (i.e., task migration) takes a certain amount of time and consumes power as the platform agent has to find a free core and has to migrate a task to that core. Moreover, the reallocation of a task to a cell which is allocated at a great distance from other cells of the application leads to drastically decreased performance since the communication cost increases.

Cluster agents functionality includes only dynamic voltage/frequency scaling down such that these parameters can only be decreased. These procedures could enhance reliability of the platform. For instance, if the temperature of a cluster exceeds some threshold, lowering frequency and/or voltage can reduce the dissipated power such that the temperature is decreased. However, applying only these procedures at the cluster level may not be efficient since the whole cluster runs at a lower frequency, which decreases the overall computational performance. Furthermore, the task reallocation procedure can only be performed by the platform agent, which may overload it.

Each cell is managed by a local cell agent. Hence, the cell agents are specified using total functions in order to represent all the cell agents in the platform. The functionality of the cell agents is modeled using non-deterministic events that had local variables. The cell agent of a faulty core initiates a local reconfiguration procedure after a task of this faulty core has been reallocated. This procedure aims at recovering the functionality of the cell, so that it is reused in the computations. Consequently, the original performance of the computations can be restored.

Overall, the specification described in [9] is developed as a monolithic formal model that includes the generic functionality of all three levels. Then, it is decomposed using the shared-variable style [23], where shared variables shown in Fig. 2 represent the interfaces between the levels, so that we can continue with further individual refinements. However, the decomposition of the model turned out to be too restrictive from the refinement point of view. More specifically, the shared structures (i.e., shared variables and external events) of a decomposed model could not be refined. Refinement of the shared structures is an ongoing research topic within the Event-B community [24]. Currently, the initial results are not yet implemented in the Rodin platform. Hence, we generated three models out of the previously developed model only keeping the details of each level (sets, axioms, variables, invariants and events) as if the model was decomposed.

3.4. Proposed approach

In this paper, we continue individual parallel refinements of all three agent levels. Firstly, we adopt an existing mapping algorithm [13] and extend it with spare cores allocation within a region for the platform agent. We provide a specific mapping function that substitutes the simplistic function defined previously. The spare cores are not running computations, i.e., they are in the idle mode, and, hence, they consume the least power. We consider task graphs that provide information about the application tasks and transitions (communication) between these tasks. When the platform agent maps an application onto a region, it provides information about tasks allocation as well as the task graph to a corresponding cluster agent.

Secondly, we propose a new algorithm for efficient utilization of these spare cores by the cluster level agents. The corresponding cluster agent, which is created initially when an application is mapped, reassigns a task of a faulty cell to a spare one considering an application task graph when a fault occurs. This allows a more efficient task reallocation in terms of speed (e.g., time consumed when searching for a free cell and actual reallocation of a task) and energy consumption than that at the platform level. Furthermore, the cluster agent restores frequency and voltage such that the computations can proceed as efficient as possible. Consequently, the architecture of the cluster agents is fully distributed, i.e., each cluster agent independently manages its region by utilizing the spare cores without overloading the platform agent.

Finally, the cell agent is typically a simplistic control device that reads the inputs and updates the outputs depending on the values of the inputs just read. It is usually implemented as a hardware unit [25] using, for example, VHDL. However, the base specification that contains total functions and non-deterministic events cannot be directly used for code generation. Therefore, we continue the development following the refinement approach such that an implementable model of the cell agent is derived and VHDL code is generated [26] from this specification.

When the desired hierarchical structure of the platform, cluster and cell agents has been derived following a stepwise and correct-by-construction formal process, the specification of the system is split into three layers for further individual development. To continue with further parallel refinements of the agents incorporating the described functionality, we propose the following design flow:

For the platform agent:

1. Adjust the mapping function in accordance with the algorithm proposed in Section 4.1.
2. Refine the model by extending it for processing task graphs that should be supplied to the platform following the description in Section 4.2.

For the cluster agents:

1. Extend the functionality such that frequency and voltage can be restored.
2. Refine the model considering task graphs and the algorithm in Section 4.3.

For the cell agent towards hardware implementation (Section 4.4):

1. Refine the specification of the cell agents by specifying coordinates of the agent and eliminating the non-determinism of events.
2. Refine the model by introducing simply typed variables and gluing invariants such that the functions are eliminated.
3. If there are variables that are involved in guards and in assignments of a specification simultaneously, these variables represent a loopback in hardware. To model a loopback, refine these variables by two simply typed variables, where one of them is an input and the other one is an output.

In the next sections, we elaborate on the proposed approach in details. We present algorithms for all three levels and show the formal development of the allocation, reallocation and reconfiguration procedures at different levels of the agents hierarchy.

4. Application mapping and reconfiguration

4.1. Application mapping with spare cores

The application domain we aim at is critical systems [2,3]. On the one hand, such systems require some redundancy in order to achieve the necessary level of reliability [6]. On the other hand, they are envisaged to be deployed on many-core platforms due to requirements on efficiency in terms of, for instance, power consumption and/or performance (e.g., [31,32]). Therefore, the design of such systems has to provide a balanced tradeoff between fault-tolerance and efficiency. This can be achieved by using spare cores available for utilization when required. Based on [19], the number of spare cores depends on different factors such as chip yield, manufacturing, service cost, etc. Nevertheless, having spare cores improves dependability.

Considering these requirements, we adopt the generic algorithm to tasks allocation presented by Chou and Marculescu [18] and extend it with spare cores allocation within a region as shown in Fig. 3, where n represents the number of requested resources and n ÷ m returns a quotient.

```
1: Find for an application a region that contains n requested resources and the calculated number of spare cores:
        if (n is 1 to 3) then
                the number of spare cores = n
                region = n rows * two columns
        else {if (n > 3)}
                the number of spare cores = (n + 1) ÷ 2
                region = (n + 1) ÷ 2 rows * three columns
        end if
2: if (the region is found ) then map the application tasks there such that :
        a) their communication is as efficient as possible [18]
        b) the rightmost column retains free spare cores
  end if
```

Fig. 3. The algorithm of initial application mapping with spare cores.

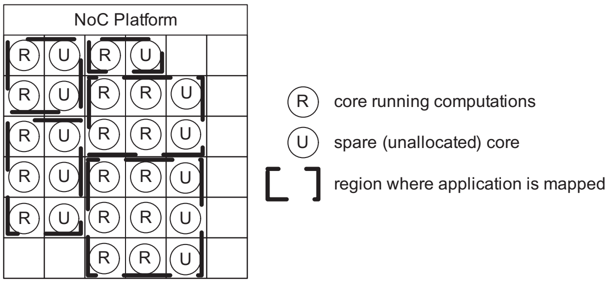


Fig. 4. Application mapping.

The platform agent performs the initial application mapping in the following manner. The application is mapped onto a rectangular shaped region that contains the rightmost column of spare cores. Depending on the number of resources requested by an application, the region with spare cores varies. If an application requests from one to three cores, the region contains the duplicated number of cores such that the number of spare cores in the rightmost column conforms to the number of requested resources. Since we target this system towards critical (safety–critical) applications and the corresponding standards [6] require redundancy for applications with high safety integrity level, we propose to allocate spare cores within a region so that their total number equals to the half of the requested number of cores for an application requiring more than three cores. Hence, the region has at most three columns, where the rightmost column contains unallocated spare cores. This allows a balanced tradeoff between fault-tolerance and efficiency for such an application (Fig. 4). In Fig. 4 and later on, the circles with "R" represent cores running computations while the circles with "U" are the spare (unallocated) cores.

When an application is mapped onto a corresponding region, the platform agent allocates application tasks inside this region considering the application task graph.

4.2. Tasks allocation within a region

A task graph (or an application characteristic graph) [18,22] is a directed graph that contains the information about vertices and transitions. The vertices specify the tasks (or groups of tasks) while the transitions show the communication between the tasks. In particular, the transitions denote the communication bandwidth between different tasks. Therefore, while processing the task graph of an application, the agents can utilize the NoC platform in an efficient manner.

The tasks allocation procedure runs in a similar manner for all cores within the region as in [18], except for the cores allocated at the rightmost column. These cores remain unallocated for future utilization by a corresponding cluster agent. The platform agent starts tasks allocation by assigning the task that communicates the most with other tasks (i.e., the task that has the most number

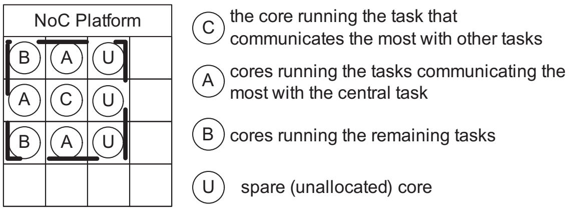

NoC Platform
B
A
U
A
C
U
B
A
U
C the core running the task that communicates the most with other tasks
A cores running the tasks communicating the most with the central task
B cores running the remaining tasks
U spare (unallocated) core

Fig. 5. Task allocation inside a rectangular region.

Table 1
Simulation of communication with the core at position 1.

Source	Destination	Average delay, cycles	Energy, Joule	Received packets
0	1	4	2.469e−09	4
2	1	5.75	2.469e−09	4
3	1	6.25	4.938e−09	4
6	1	7.75	4.938e−09	4
8	1	9.25	7.407e−09	4
15	1	12	1.234e−08	4

Table 2
Simulation of communication with the core at position 0.

Source	Destination	Average delay, cycles	Energy, Joule	Received packets
1	0	5	2.469e−09	7
2	0	9.85714	4.938e−09	7
3	0	9.42857	7.407e−09	7
6	0	10.2857	7.407e−09	7
9	0	8.71429	7.407e−09	7
15	0	21.7143	1.481e−08	7

of links in the task graph) to a core in the center of the region. Then, it proceeds with allocating tasks that communicate with this central task with the highest bandwidth to the cores around the central one. Finally, it assigns the remaining tasks to the free cores that are left after the central task and the most communicating tasks are allocated. An example of application mapping is shown in Fig. 5, where the circle with "C" illustrates the core running the task that communicates the most with other tasks (i.e., the central task/core), the circles with "A" show the tasks around the central one while the circles with "B" are the remaining tasks assigned to the corresponding cores of the region.

After the platform agent completes an application mapping and tasks allocation, it creates a cluster agent for the region and provides it with information about tasks allocation as well as the application task graph. This scheme retains the simplicity of a cluster agent as well as provides an efficient fault-tolerance mechanism, namely a task migration within the region as the corresponding cluster agent is aware of having spare cores and the application task graph. The task migration inside the region is deterministic.

4.3. Task reallocation inside a region

A corresponding cluster agent performs task reallocation inside a region to one of the closest spare cores. This procedure consumes less energy and time than reassigning a task to a free core outside the region in the platform. Moreover, the efficiency of the result of this procedure is directly affected by the communication performance of the underlying platform. Hence, the communication performance between the tasks within the region remains at an adequate level.

To justify the above arguments, let us consider an example of the application mapping that is shown in Fig. 6 and elaborate on simulation results provided by the Noxsim simulator [27]. The simulation results have been chosen in such a manner that the number of packets going from source to destination of interest is the same and with minimum delay. This is important for the results to be comparable. In Fig. 6, the circles with "F" illustrate the faulty cores and the circles with "S" show their substitutions.

Suppose the core at position 0 sends four packets to the core at position 1. Whenever the core fails, the cluster agent can reallocate a task within the region either to the core at position 2 or to the core at position 6 depending on the availability of these cores. The platform agent, in its turn, can reallocate a task from this core to any core in the platform which is free, i.e., no application is mapped to that core. For this example, we consider the cores at positions 3, 8 as the closest ones and at position 15 in the worst case (Fig. 6a). The simulation results for these cases (Table 1) show that the communication cost after task migration within the region is lower than that of within the platform.

Similarly, consider another example where the core at position 1 sends seven packets to the core at position 0. The substituting cores are at positions 2 or 6 (within the region) and 3, 9 or 15 (outside the region) as shown in Fig. 6b. The simulation results for this case are shown in Table 2.

Please note that the example considers a single application mapped onto the platform, i.e., the cores outside the region where the application is mapped can be used for the task migration. However, this might not be possible due to many applications running in the system. Moreover, the routers of the platform perform inter-region communications between other cores affecting the delays and energy consumption shown in the tables. For the work in this paper, we do not restrict the usual routing of the NoC platform.

Considering the results presented in the tables, their comparison shows that the farther a task is migrated the more cycles (higher delay) and energy are required for a packet to reach the destination. Hence, having spare cores within the region enhances efficient utilization of the platform while allowing it to reach the required level of reliability.

The task reallocation procedure allows an application to continue its execution without interruption. For efficient task reallocation within a region, we propose a new algorithm shown in Fig. 7.

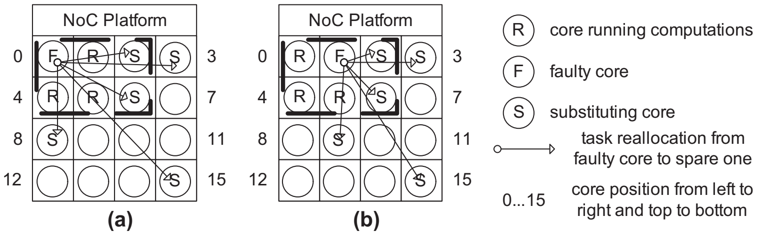

NoC Platform
0
F
R
S
S
3
4
R
R
S
7
8
S
11
12
S
15
(a)
NoC Platform
0
R
F
S
S
3
4
R
R
S
7
8
S
11
12
S
15
(b)
R core running computations
F faulty core
S substituting core
task reallocation from faulty core to spare one
0...15 core position from left to right and top to bottom

Fig. 6. Examples of task reallocation within the cluster and within the platform.

```
1: if (a fault occurs in a topmost cell) then
        if (the spare cell allocated on a row below is free) then reallocate task there
        else if (the spare cell allocated on the same row is free) then reallocate task there
        else reallocate task to any available free spare core, if any end if
2: if (a fault occurs in a bottommost cell) then
        if (the spare cell allocated on a row above is free) then reallocate task there
        else if (the spare cell allocated on the same row is free) then reallocate task there
        else reallocate data to any available free spare core, if any end if
3: if (a fault occurs in a cell which is not topmost nor bottommost) then
        if (the spare cell allocated on the same row is free) then reallocate task there
        else if (there are free spare cores on a row below and on a row above) and
            (there are transitions with the tasks allocated on a row above or below) then
              if (the number of transitions with the task on a row below > the number of transitions with the task on a row above) then
                    reallocate task to the spare core allocated on a row below
              else if (the number of transitions with the task on a row above > the number of transitions with the task on a row below) then
                    reallocate task to the spare core allocated on a row above
              else
              {the number of transitions with these tasks is the same}
                    if (the bandwidth with the upper task ≥ the bandwidth with the lower task) then
                          reallocate task to a spare core on a row above
                    else reallocate task to a spare core on a row below end if
              end if
        else
        {the spare cores allocated on a row below, the same row and a row above are taken) Or}
        {there are tasks on a row below and above, but there are no transitions to them}
              reallocate data to any available free spare core, if any
  end if
```

Fig. 7. The algorithm of task migration performed by a cluster agent.

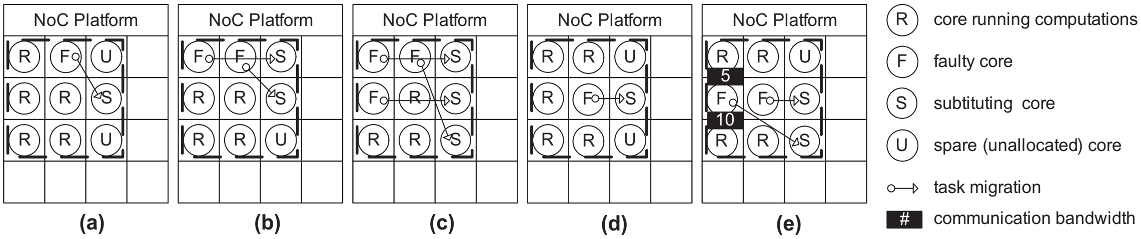


Fig. 8. Task migration within a region.

The algorithm does not depend on the size of the platform since it is performed within a cluster. Hence, it is applicable to large-scale platforms.

The algorithm reallocates a task within the region such that the communication cost is minimum. In general, there are three cases for a task migration inside a region. The first one considers faults that occur in a top-most cell. According to the algorithm of the initial task allocation which assigns tasks in a stepwise and incremental manner starting from the central task, a top-most core most probably runs a task that has high communication bandwidth with a task assigned to a core on a row below. Hence, the cluster agent first tries to move the task from this cell to a spare one which is on the row below so that these tasks are still allocated as close as possible. For instance, in Fig. 8a) a fault occurs in a top-most cell and a task is reallocated to the spare core in the middle so that the communication distance between the reallocated task and the task running on the core in the middle of the region remains the same. If this core has been already allocated, the cluster agent attempts to “mirror” the task, i.e., it reassigns the task from a faulty core to a spare one which is on the same row as the faulty core (Fig. 8b). Finally, if none of these cores are free, the cluster agent migrates the task to any core that is available in the set of spare cores (Fig. 8c). In a similar manner, the cluster agent performs task reallocation for faults occurring in a bottom-most core.

Finally, a fault can occur in a core which is neither top-most nor bottom-most, i.e., the core is in the middle of a region. In this case, the cluster agent reassigns the task from this cell to a spare cell on the same row, if this cell is free (Fig. 8d). If this cell is allocated, then the cluster agent is required to process the application task graph [18], i.e., the transitions of the task allocated on the faulty core. In particular, it inspects the number of links as well as the bandwidth between this task and the tasks one row up and one row down. Depending on the analysis results, the cluster agent reallocates the task to a spare core with the higher communication bandwidth (Fig. 8e). Lastly, if none of these cores are free, the cluster agent moves a task to any available free spare core.

Whenever a cluster agent reallocates a task from a faulty core to a spare one, it keeps track between them. This allows the cluster agent to move the task back when the local reconfiguration procedure is complete. The local reconfiguration procedure starts when the task has been reallocated. A corresponding cell agent applies the reconfiguration command to the faulty cell. This procedure aims at recovering the functionality of the cell such that this cell is reused in the computations. Therefore, the original performance is restored.

4.4. Local reconfiguration

For the local reconfiguration of a cell, we adopt the functionality of the cell agents we proposed in [9]. The algorithm performed by the cell agents is shown in Fig. 9. The reconfiguration of a cell at the hardware level stands for modifying the internal structure of it as if it was, for instance, a single FPGA chip [28,29]. The modification is performed via uploading a new configuration file to a core.

5. Formal modeling of the platform agent

Before we present the formal development within the Event-B formalism, we show the summary of used symbols in Table 3.

```
while (true) do
        1: monitor the state of a cell
        2: promote data about the current state of the cell to the cluster agent
        3: if (cell is not faulty ) then continue
           else while (task is not reallocated ) do wait end while
                initiate cell reconfiguration
                while (the cell is not reconfigured ) do wait end while
           end if
end while
```

Fig. 9. The algorithm of the cell agent.

For the complete detailed formal definitions of the Event-B notations, the reader is referred to [30].

5.1. The platform agent: application mapping with spare cores

To proceed with the development of the platform agent taking into account the proposed algorithm (Section 4.1) for the initial application mapping, we introduce the function that returns a rectangular shaped region with spare cores, namely mapfun. The function takes three arguments (the number of requested resources and the current position of indices in the matrix) and returns a region. Since a region where an application is mapped has at most three columns (see Fig. 3), one of them is reserved for spare cores:

$$\textbf{axiom}\ \mathsf{mapfun} \in 1..2 * \mathsf{IPnum} \times \mathsf{NoC} ⇸ \mathbb{P}1(\mathsf{NoC})$$

$$\textbf{axiom}\ \mathsf{partition}(\mathsf{dom}(\mathsf{mapfun}), \{(n\mapsto(x\mapsto y))|n \in 1..3 \wedge x \in 1..\mathsf{IPnum} - (n-1) \wedge y \in 1..\mathsf{IPnum} - 1\}, \{(n\mapsto(x\mapsto y))|n \in 4..2 * \mathsf{IPnum} \wedge x \in 1..\mathsf{IPnum} - ((n+1) \div 2 - 1) \wedge y \in 1..\mathsf{IPnum} - 2\})$$

where $\mathsf{NoC} = 1...\mathsf{IPnum} \times 1...\mathsf{IPnum}$ stands for a matrix of cores (i.e., a NoC platform) assumed to be a 2D mesh of square shape, $\mathsf{IPnum} \geqslant 2$ is the constant defining the size of this matrix, n represents the number of requested resources and a pair $x\mapsto y$ reflects the coordinates in the matrix. Note however that the same approach can be applied to other topologies and dimensions since the approach is independent of the routing scheme.

The definition of this function consists of two axioms stating the result the function returns depending on its arguments. The first axiom postulates that if an application requests from 1 to 3 cores, the result is a rectangular region that contains two columns and the number of rows that conforms to the number of requested resources:

$$\textbf{axiom}\ \forall n, x, y.\ n \in 1..3 \wedge x \in 1..\mathsf{IPnum} - (n-1) \wedge y \in 1..\mathsf{IPnum} - 1 \Rightarrow \mathsf{mapfun}(n\mapsto(x\mapsto y)) = x..x + (n-1) \times y..y + 1$$

The second axiom is for the case where an application requests 4 or more cores. Due to the fact that the division operation (denoted by $\div$) within Event-B is the integer division ($\forall k.\ k \in \mathbb{N} \Rightarrow (k+1) \div 2 = \lceil k/2 \rceil$), the mapping function always returns a rectangular shaped region that contains the rightmost column of spare cores (see Fig. 3):

$$\textbf{axiom}\ \forall n, x, y.\ n \in 4..2 * \mathsf{IPnum} \wedge x \in 1..\mathsf{IPnum} - ((n+1) \div 2 - 1) \wedge y \in 1..\mathsf{IPnum} - 2 \Rightarrow \mathsf{mapfun}(n\mapsto(x\mapsto y)) = x..x + ((n+1) \div 2 - 1) \times y..y + 2$$

The mapping of an application onto an NoC platform proceeds similarly to the mapping described in [9]. From now on, we only show the variables and the parts of the events that have been affected by the specified mapping with spare cores. Complete events can be found in Appendix A.

Application requests are stored as a partial function that maps an application to the number of requested resources. An application can request a number of resources that ranges between 1 and 2*IPnum:

$$\textbf{invariant}\ \mathsf{pending\_apps} \in \mathsf{APPLICATIONS} ⇸ 1..2*\mathsf{IPnum}$$

The request of resources is modeled with the event Request_resources that has been derived from the base model. This event has also been updated according to this requirement:

Table 3
Summary of used formal symbols.

Symbol	Description
$\varnothing$	The empty set
$\mathbb{P}(S)$	The power set of set S
$\mathbb{P}1(S)$	$\mathbb{P}(S) \setminus \{\varnothing\}$
card(S)	Cardinality (i.e., the number of elements) of set S
partition(S,A,B)	Enumerated set comprehension such that $S = A \cup B$ and $A \cap B = \varnothing$
finite(S)	Specifies that the set S is finite
n..m	An interval, i.e., the set of numbers starting from n and ending in m
min(S)	A minimum element of the set S, where $S \subset \mathbb{Z}$ and must have a lower bound
max(S)	A maximum element of the set S, where $S \subset \mathbb{Z}$ and must have an upper bound
$x\mapsto y$	An ordered pair
$X \times Y$	Cartesian product of X and Y, i.e., the set of all possible ordered pairs where the first entry belongs to X and the second entry belongs to Y
id	An identity, i.e., the set of ordered pairs whose both entries are the same
$\mathsf{dom}(f) \subseteq S$	The domain of a relation f
$\mathsf{ran}(f) \subseteq T$	The range of a relation f
$f \in S ⇸ T$	A partial function from set S to set T
$f \in S \rightarrow T$	A total function (dom(f) = S) from set S to set T
$f \in S ⤔ T$	A partial injective (one-to-one) function from set S to set T
$f \in S ⤀ T$	A partial surjective (ran(f) = T) function from set S to set T
$f \in S \twoheadrightarrow T$	A total surjective (dom(f) = S and ran(f) = T) function from set S to set T
$f \rhd R$	Range restriction of the relation f by the set R
$f ⩤ R$	Domain subtraction from the relation f the set R
$f O$	Relational override of the relation f with the set O
f;g	Forward composition of the relations f and g

```
r, c ← 1, 1
while not (region is found) do
    if (indices are at maximums) then
        reset indices so that a new search procedure is initiated
        exit
    end if
    if (n is 1 to 3) then
        if (c < IPnum - 1) then
            c ← c + 1
        else if (r < IPnum - (n - 1)) then
            r, c ← r + 1, 1
        end if
    else {n > 3 and n <= 2*IPnum}
        if (c < IPnum - 2) then
            c ← c + 1
        else if (r < IPnum - ((n + 1) ÷ 2 - 1)) then
            r, c ← r + 1, 1
        end if
    end if
end while
```

Fig. 10. The region search algorithm.

event Request_resources $\triangleq$ **any** app res_num
where
// There is an application that wants to run computations
$\mathrm{app} \in \mathrm{APPLICATIONS} \wedge \mathrm{app} \notin \mathrm{dom(pending\_apps)} \wedge$
$\mathrm{app} \notin \mathrm{ran(mapping)} \wedge$
// The number of requested resources is in-between 1 and 2∗IPnum
$\mathrm{res\_num} \in 1..2*\mathrm{IPnum}$
then
$\mathrm{pending\_apps} := \mathrm{pending\_apps} \cup \{\mathrm{app} \mapsto \mathrm{res\_num}\}$
end

The platform agent seeks for resources in the platform in a linear manner. It attempts to find a region for an application starting with the top-most and leftmost cell whose coordinates are 1, 1. Then, it proceeds throughout columns and rows incrementally according to the algorithm shown in Fig. 10, where n is the number of requested resources while r and c are the row and the column indices in the matrix, respectively. In case the searching procedure is unsuccessful, the platform agent resets the indices ($r = 1 \wedge c = 1$) such that a new searching procedure can be initiated.

The searching procedure is modeled using convergent events. The convergence of the events along with the corresponding variant (PO (VAR) in Section 3.1) guarantees that the platform agent can always find a region for an application, if such a region exists. The shape of the region remains consistent (i.e., of the rectangular shape) according to the algorithm presented in Fig. 3 (Section 4.1), if no task reallocation occurs at the platform agent level. This is ensured by the following invariance property:

invariant $\forall a.a \in \mathrm{ran(mapping)}$
$$\begin{aligned}
&\Rightarrow ((\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \cap \mathrm{ran(Cell\_trace)} = \varnothing)\\
&\Rightarrow \mathrm{dom}(\mathrm{mapping} \rhd \{a\})\\
&= \min(\mathrm{dom}(\mathrm{dom}(\mathrm{mapping}\\
&\quad \rhd \{a\})))..\max(\mathrm{dom}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})))\\
&\quad \times \min(\mathrm{ran}(\mathrm{dom}(\mathrm{mapping}\\
&\quad \rhd \{a\})))..\max(\mathrm{ran}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\}))))
\end{aligned}$$

where $\mathrm{mapping} \in \mathrm{NoC} \twoheadrightarrow \mathrm{running\_apps}$ is the model variable that stores the mapping between cores (their coordinates) and running applications ($\mathrm{running\_apps} \subseteq \mathrm{APPLICATIONS}$), $\mathrm{Cell\_trace} \in \mathrm{NoC} \setminus \mathrm{dom(mapping)} \rightarrowtail \mathrm{dom(mapping)}$ stores the track of task reallocation performed by the platform agent and the antecedent $\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \cap \mathrm{ran(Cell\_trace)} = \varnothing$ specifies that no task reallocation has been performed at the platform agent level.

After the resources have been found, i.e., there is a rectangular shaped region that satisfies the application request, the platform agent assigns the application to the found region. In other words, the platform agent stores the connection between the found region and the application:

event Resources_found $\triangleq$ **any** app
where $\ldots \wedge$
$(\mathrm{pending\_apps(app)} \in 1..3 \Rightarrow$
$r < \mathrm{IPnum}-(\mathrm{pending\_apps(app)}-1) \wedge c = \mathrm{IPnum}-1) \wedge$
$(\mathrm{pending\_apps(app)} \in 4..2*\mathrm{IPnum} \Rightarrow r <$
$\mathrm{IPnum}-((\mathrm{pending\_apps(app)}+1)\div 2-1) \wedge c = \mathrm{IPnum}-2)$
then $\ldots \;||\; \mathrm{mapping} := \mathrm{mapping} \cup$
$(\mathrm{mapfun}(\mathrm{pending\_apps(app)} \mapsto (r \mapsto c)) \times \{\mathrm{app}\})$
end

where $r \in 1..\mathrm{IPnum}$ and $c \in 1..\mathrm{IPnum}$ are the model variables specifying the row and the column indices in the NoC matrix, respectively.

In the next refinement, we introduce the actual task allocation procedure with spare cores for the platform agent. We consider application task graphs that allow the platform agent efficiently assign tasks to cores.

5.2. The platform agent: tasks allocation considering task graphs

In general, applications are characterized by task graphs [18,22]. We assume that these task graphs are provided to the platform. Formally, a task graph is a directed graph and is denoted as a tuple $G = (V, T)$, where V is a set of vertices that determine tasks (or groups of tasks) and T is a set of transitions with specified bandwidth between the tasks. Every application is defined by a finite graph. Hence, the sets of vertices and transitions are finite.

To represent task graphs in Event-B, we first introduce the sets of vertices and transitions into a new context. The complete context can be found in Appendix A. The generic deferred set of all vertices that applications consist of is shown below:

sets VERTICES

axiom finite(VERTICES)

Transitions represent a partial function that maps a pair of vertices to some positive number. This number shows the bandwidth between tasks in a task graph:

constant TRANSITIONS
axiom $\mathrm{TRANSITIONS} \in \mathrm{VERTICES} \times \mathrm{VERTICES} \nrightarrow \mathbb{N}1$

Since the vertices determine the tasks of the application, they cannot communicate with themselves. In other words, self-transitions are implemented internally and the tasks communicate through the NoC only with other tasks. Hence, the self-transitions are not allowed:

axiom $\mathrm{id} \cap \mathrm{dom(TRANSITIONS)} = \varnothing$

To specify the relation between the deferred set of applications defined previously [9] and task graphs, we introduce two functions: one that stores a set of vertices and the other one that returns a set of transitions for a particular application, namely app_verts and app_trans, respectively:

axiom $\mathrm{app\_verts} \in \mathrm{VERTICES} \twoheadrightarrow \mathrm{APPLICATIONS}$

axiom $\mathrm{app\_trans} \in \mathrm{APPLICATIONS} \rightarrow \mathbb{P}(\mathrm{TRANSITIONS})$

Notice that the function app_trans maps applications that consist of a single vertex to the empty set since such an application does not have transitions in its task graph. Otherwise, the function returns a set of transitions for a particular application.

The properties of these functions are postulated as a number of axioms. The first axiom states that if an application task graph contains at least two vertices, there must be a transition between them:

$$\textbf{axiom}\ \forall a, v.\ a \in \mathrm{APPLICATIONS} \land (\mathrm{card}(\mathrm{app\_verts} \rhd \{a\}) \geqslant 2) \land v \in \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\}) \Rightarrow (\exists v'.\ v' \in \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\}) \land \neg(v = v') \land ((v \mapsto v') \in \mathrm{dom}(\mathrm{app\_trans}(a)) \lor (v' \mapsto v) \in \mathrm{dom}(\mathrm{app\_trans}(a))))$$

On the other hand, transitions of a particular task graph must only have those vertices that belong to this task graph. Therefore, if a vertex does not belong to the application task graph, there is no transition to this vertex. This property is specified by the axiom shown below:

$$\textbf{axiom}\ \forall a, v.\ a \in \mathrm{APPLICATIONS} \land v \in \mathrm{VERTICES} \land \neg(v \in \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\})) \Rightarrow \neg(v \in \mathrm{dom}(\mathrm{dom}(\mathrm{app\_trans}(a)))) \land \neg(v \in \mathrm{ran}(\mathrm{dom}(\mathrm{app\_trans}(a))))$$

Finally, distinct applications have disjoint sets of transitions. This property is postulated as follows:

$$\textbf{axiom}\ \forall a1, a2.\ a1 \in \mathrm{APPLICATIONS} \land a2 \in \mathrm{APPLICATIONS} \land \neg(a1 = a2) \Rightarrow (\mathrm{app\_trans}(a1) \cap \mathrm{app\_trans}(a2) = \varnothing).$$

After defining necessary constants and functions, we proceed with the task allocation functionality at the platform level. For this purpose, we refine the previous model of the platform agent. We start by introducing several variables modeling a continuous procedure of tasks allocation within a region. Firstly, when the platform agent finds an appropriate region for an application, it has to store the application task graph for further processing. As the application task graph represents a pair, two variables are required. The first one stores the unallocated vertices the application task graph has:

$$\textbf{invariant}\ \mathrm{app\_vertices} \in \mathrm{ran}(\mathrm{mapping}) \to \mathbb{P}(\mathrm{VERTICES}).$$

Note that this variable maps an application to a power set of vertices including the empty set. When assigning a task (a vertex) to a core, the platform agent removes the task from this variable. Consequently, the platform agent identifies that all the tasks of the application have been allocated when this application is mapped to the empty set.

The second variable stores the transitions of the application task graph, if any, so that the platform agent can process them as well:

$$\textbf{invariant}\ \mathrm{app\_transitions} \in \mathrm{ran}(\mathrm{mapping}) \to \mathbb{P}(\mathrm{TRANSITIONS})$$

Secondly, the platform agent has to store the actual location of application tasks. It uses another two variables. The first one specifies the central core to which the most communicating task of the application is assigned. If there are several tasks that have the same maximum number of communication links, the platform agent nondeterministically chooses one of them. This task is the starting point for allocating other tasks of the application. Furthermore, this task must be allocated within the region where the application is mapped:

$$\textbf{invariant}\ \mathrm{app\_ctallocated} \in \mathrm{ran}(\mathrm{mapping}) \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{dom}(\mathrm{mapping})$$

$$\textbf{invariant}\ \forall a\ .\ a \in \mathrm{dom}(\mathrm{app\_ctallocated}) \Rightarrow \mathrm{app\_ctallocated}(a) \in \mathrm{dom}(\mathrm{mapping} \rhd \{a\})$$

The second variable stores the mapping between the locations (i.e., the cores) and the tasks. This variable is an injective function meaning that only one vertex (one task or one group of tasks) can be allocated to one cell (core). In addition, whenever the application task is allocated to a core, it must be assigned to a core within the region that belongs to this application:

$$\textbf{invariant}\ \mathrm{app\_tasks\_allocation} \in \mathrm{dom}(\mathrm{mapping}) \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{VERTICES}$$

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{ran}(\mathrm{mapping}) \Rightarrow (\forall x.\ x \in \mathrm{dom}(\mathrm{app\_tasks\_allocation}; (\mathrm{app\_verts} \rhd \{a\})) \Rightarrow x \in \mathrm{dom}(\mathrm{mapping} \rhd \{a\}))$$

After introducing necessary variables to model task allocation, we postulate several invariant properties that must hold for the whole model. Here, we only show the essential properties that are crucial for the task allocation procedure.

The task allocation procedure has an iterative behavior. The platform agent analyses the transitions and assigns tasks to cores in a stepwise manner. This process ends when the least communicating task is allocated. However, whenever the platform agent allocates tasks, it has to keep the rightmost column of spare cores unallocated according to the algorithm presented in Fig. 3 (Section 4.1). This main functional property is stated as the following invariant:

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{ran}(\mathrm{mapping}) \Rightarrow (\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \cap \mathrm{ran}(\mathrm{Cell\_trace}) = \varnothing \land ((\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \cap \mathrm{ran}(\mathrm{Cluster\_Cell\_Trace}) = \varnothing) \Rightarrow (\mathrm{dom}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})) \times \{\max(\mathrm{ran}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})))\}) \cap \mathrm{dom}(\mathrm{app\_tasks\_allocation}) = \varnothing)$$

where $\mathrm{Cluster\_Cell\_Trace} \in \mathrm{dom}(\mathrm{mapping}) \rightarrowtail\!\!\!\!\twoheadrightarrow \mathrm{dom}(\mathrm{mapping})$ is the track of task reallocation executed by a cluster agent (Section 6). The premise $((\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \cap \mathrm{ran}(\mathrm{Cluster\_Cell\_Trace}) = \varnothing)$ reflects that no task has been reallocated within the region, i.e., the region remains consistent being of the rectangular shape determined by the mapping function. The value $\max(\mathrm{ran}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})))$ represents the rightmost column in the region.

To be able to verify the main property shown above, we introduce invariants that determine the relationship between variables storing task graphs and application (tasks) mapping. Firstly, since every application is defined by a unique task graph, the sets of vertices of different applications must be disjoint. For the same reason, the sets of transitions of different applications must be disjoint as well:

$$\textbf{invariant}\ \forall app1, app2.\ app1 \in \mathrm{dom}(\mathrm{app\_vertices}) \land app2 \in \mathrm{dom}(\mathrm{app\_vertices}) \land \neg(app1 = app2) \Rightarrow \mathrm{app\_vertices}(app1) \cap \mathrm{app\_vertices}(app2) = \varnothing$$

$$\textbf{invariant}\ \forall app1, app2.\ app1 \in \mathrm{dom}(\mathrm{app\_transitions}) \land app2 \in \mathrm{dom}(\mathrm{app\_transitions}) \land \neg(app1 = app2) \Rightarrow \mathrm{app\_transitions}(app1) \cap \mathrm{app\_transitions}(app2) = \varnothing$$

Secondly, when the platform agent has found a region, it has to store the exact task graph of an application. In particular, if the platform agent has not assigned the central task to a core within the region, the variables app_vertices and app_transitions applied to a particular application conform to the variables app_verts and app_trans restricted to the same application, respectively. Furthermore, if the central task has not been assigned to a core, none of the tasks have been allocated either:

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{dom}(\mathrm{app\_vertices}) \wedge \neg(a \in \mathrm{dom}(\mathrm{app\_ctallocated})) \Rightarrow \mathrm{app\_vertices}(a) = \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\}) \wedge \mathrm{app\_transitions}(a) = \mathrm{app\_trans}(a)$$

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{ran}(\mathrm{mapping}) \wedge \neg(a \in \mathrm{dom}(\mathrm{app\_ctallocated})) \Rightarrow (\forall v.\ v \in \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\}) \Rightarrow \neg(v \in \mathrm{ran}(\mathrm{app\_tasks\_allocation}))).$$

When allocating a task to a core, the platform agent removes this task from the previously stored set of vertices. Therefore, if there is a vertex in the set of vertices, this vertex (this task) has not been allocated to a core in the platform:

$$\textbf{invariant}\ \forall a, x.\ a \in \mathrm{dom}(\mathrm{app\_vertices}) \wedge x \in \mathrm{app\_vertices}(a) \Rightarrow \neg x \in \mathrm{ran}(\mathrm{app\_tasks\_allocation})$$

Finally, the task allocation procedure is completed when an application is mapped to the empty set in the variable app_vertices, i.e., all the application tasks are allocated. Hence, the function app_tasks_allocation applied to the region where the application is mapped returns the exact set of vertices that belong to this application:

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{ran}(\mathrm{mapping}) \Rightarrow (\mathrm{app\_vertices}(a) = \varnothing \iff \mathrm{app\_tasks\_allocation}[\mathrm{dom}(\mathrm{mapping} \rhd \{a\})] = \mathrm{dom}(\mathrm{app\_verts} \rhd \{a\})).$$

Next, we show the refined parts of the abstract events and provide new events that model the task allocation procedure. The events implement the procedure described in Section 4. The complete formal representation of the events can be found in Appendix A.

In order for the model to be consistent with the stated invariants, we refine the event Resources_found. We specify that when the platform agent has found the region for an application, it stores the application task graph such that the task allocation procedure within the region can be initiated:

```
event Resources_found ≙ any app
where ... ∧
  // If there is a free region where an application can be
   mapped, i.e., the region is found
  // Choose the application that has not been processed yet
  $\neg$// $\mathrm{dom}(\mathrm{app\_verts} \rhd \{\mathrm{app}\}) \in \mathrm{ran}(\mathrm{app\_vertices}) \wedge$
  // And its tasks are not allocated yet ∧
  $\mathrm{dom}(\mathrm{app\_verts} \rhd \{\mathrm{app}\}) \cap \mathrm{ran}(\mathrm{app\_tasks\_allocation}) = \varnothing$
then ... ||
  // Create the cluster agent for the region
  // Store the task graph, namely the vertices and the
   transitions
  $\mathrm{app\_vertices} := \mathrm{app\_vertices} \cup \{\mathrm{app} \mapsto \mathrm{dom}(\mathrm{app\_verts} \rhd \{\mathrm{app}\})\}$ ||
  $\mathrm{app\_transitions} := \mathrm{app\_transitions} \cup \{\mathrm{app} \mapsto \mathrm{app\_trans}(\mathrm{app})\}$
end
```

The platform agent starts processing the task graph starting by calculating the central task. To find the central task, the platform agent examines the transitions and chooses the task that has the highest number of them. This task is assigned to a core located in the middle of the region. Since the region is of a rectangular shape, the middle of it is an arithmetic mean for both coordinates.

After the central task is allocated, the platform agent allocates all other tasks, if any. Due to the fact that the application task graph is directed, there are at most two transitions between two different tasks: one in one direction and the other one in the other direction. Therefore, the platform agent should consider both possibilities of tasks communication.

The following event models the task allocation of a vertex v that has a transition from v to some task vl already assigned to a core. Furthermore, the platform agent chooses such a vertex v that has the highest bandwidth with the vertex vl and allocates it to the coordinates rt, ct:

```
convergent event Task_allocation_Transition_to_allocated_task ≙ any app v vl
   rt ct sur
where ... ∧ // After the central task allocation is complete,
  // Choose a task that has a transition from a task being allocated to an already
   allocated task
  $(v \mapsto vl) \in \mathrm{dom}(\mathrm{app\_transitions}(\mathrm{app})) \wedge$
  // Furthermore, choose one with the highest bandwidth
  $(\forall v'.v' \in \mathrm{app\_vertices}(\mathrm{app}) \wedge \neg(v' = v) \wedge (v' \mapsto vl) \in \mathrm{dom}(\mathrm{app\_transitions}(\mathrm{app})) \Rightarrow$
  $\mathrm{app\_transitions}(\mathrm{app})(v \mapsto vl) \geqslant \mathrm{app\_transitions}(\mathrm{app})(v' \mapsto vl))$
  // Choose proper coordinates for this task within the region
  // This core must not be allocated
  // Choose coordinates near the allocated core, if any
then
  // Store the task allocation and Remove a processed transition and a
   processed task
end
```

where sur is a set of all free cores around an allocated task. It contains coordinates of the cores whose place in the same column one row above and one row below as well as the cores whose place is in the same row, but one column to the left, excluding the rows and columns outside the application region. The platform agent uses this set when there are no available cores around an allocated task, whereas there are free cores in the region and there are tasks to be allocated. In this case, the platform agent assigns such a task to a free core in the region non-deterministically.

Similarly, the platform agent checks the other possibility of tasks communication from vertex vl (an already allocated task) to v (a task being allocated). When the platform agent assigns a task to a core, it removes this task from the set of unallocated tasks. These two events are applied repeatedly until all the tasks in the task graph are allocated, i.e., until the variable app_vertices applied to the application app returns an empty set.

Due to the fact that these events as well as the event modeling allocation of the central task remove an element from a set, they must eventually terminate, so that the platform agent is able to execute other functions. Therefore, these events are convergent and the provided variant guarantees their termination (PO (VAR) Section 3.1).

The platform agent can reallocate the whole application, after all application tasks have been allocated and there is a fault in the region. This occurs when there are no free spare cores left in the cluster and there is a new fault. In this case, the platform releases the application from the platform and initiates a new search for resources. This case is modeled by event Reallocate_app whose refined parts contain the guards and the actions similar to event Computations_over. Additionally, this event has a guard specifying that all of the spare cores within the region are allocated: card(dom(mapping ▷ {app}) ∩ ran(Cluster_Cell_Trace)) = (resources+1)÷2.

Due to the fact that a cluster agent can reallocate a task from a cell to another cell, this also affects the global reallocation procedures at the platform level. Firstly, the task being reallocated must not be the central one since the reallocation of the central task drastically reduces performance of computations as the communication cost significantly increases. Secondly, whenever the platform agent reallocates the task from a faulty cell inside a region to a free cell outside the region in the platform, the shape of the cluster changes and a corresponding cluster agent is adjusted to the new topology. Finally, the platform agent can initiate reallocation of a task outside the application region only if all the spare cores within the cluster are utilized. Therefore, the events that model task migration within the platform are refined as well. For instance, the refined part of the event Reallocate_a_cell is shown below:

```
event Reallocate_a_cell refines Reallocate_a_cell ≙ any x y k l
where . . . ∧
  //Application tasks are allocated and the task being reallocated does not belong to the central task as it will
  //drastically decrease communication performance of the application
  app_vertices(app) = ∅ ∧ app ∈ dom(app_ctallocated) ∧ ¬app_ctallocated(app) = (k↦l) ∧
  //There is a free spare core in the platform where the task can be reallocated
  ¬x↦y ∈ dom(app_tasks_allocation) ∧ k ↦l ∈ dom(app_tasks_allocation) ∧
  //A cell whose task is being reallocated does not belong to the rightmost column
  ran(Cluster_Cell_Trace)∩{k ↦ l}=∅ ∧
  //The ''global'' reallocation can take place if no spare cores are left within the cluster
  card(dom(mapping ▷{app}) ∩ ran(Cluster_Cell_Trace)) =
  (card(min(dom(dom(mapping ▷ {app})))...max(dom(dom(mapping ▷ {app})))×
  min(ran(dom(mapping ▷ {app})))...max(ran(dom(mapping ▷ {app})))−1)+1)÷2 ∧
  //The global reallocation procedure modifies the shape of the region such that it is no more rectangular
  ¬dom({k↦l} ⩤ mapping ▷ {app}) ∪ {x↦y} = dom(dom({k↦l}⩤ mapping ▷ {app}) ∪ {x↦y})×
  ran(dom({k↦l}⩤ mapping ▷ {app}) ∪ {x↦y})
//Move task to that spare core
then ...|| app_tasks_allocation := {k↦l} ⩤(app_tasks_allocation ∪ {x↦y↦app_tasks_allocation(k↦l)})
end
```

Upon deriving the specification of the platform agent, we have generated and discharged proof obligations using the Rodin platform [8]. The proof statistics for this model including contexts, the abstract machine and the refinement is summarized in Table 4. From the table, we observe that the Rodin platform generates 509 proof obligations and automatically discharges 356 of them (more than 65%).

While discharging proof obligations invariant preservation and feasibility ((INV) and (FIS) in Section 3.1) interactively, we observe that these proof obligations required either case distinction technique or instantiation of quantified (bound) variables, which are relatively complex to be proved automatically. The other proof obligations ((VAR), (GRD) and (SIM) Section 3.1) for this specification were proven automatically.

Table 4
The proof statistics for the platform agent.

Model	Number of proof obligations	Automatically discharged	Interactively discharged
Contexts	9	8	1
The base machine	240	184	56
The refinement	260	164	96
Total	509	356	153

Since we consider reallocation procedures inside the region, we have also refined the specification of the cluster agents. Let us now examine the essential parts of this model.

6. Formal modeling of the cluster agents

6.1. The cluster agents: frequency and voltage restore

In the previous work [9], we have developed such a model of the cluster agents that decreased frequency and voltage within the region without eventually increasing them. However, when the task of a faulty cell is reallocated, cluster agent can restore the value of these parameters to their maximums so that the computations can proceed as efficiently as possible. Therefore, we extend the functionality of the cluster agents with a possibility of restoring the values of frequency and voltage, when required:

```
event Restore_cluster ≙ any app
where app ∈ ran(mapping)
then
Cluster_Frequency := Cluster_Frequency ⊲− {dom(mapping ▷
   {app})↦Max_Freq} ||
Cluster_Voltage := Cluster_Voltage ⊲− {dom(mapping ▷
   {app})↦Max_Volt}
end
```

where Cluster_Frequency ∈ {x | ∃a. a ∈ ran(mapping) ∧ x=dom(mapping ▷ {a})} → 0..Max_Freq is the frequency that the set of cores (the cluster) runs at and Cluster_Voltage ∈ dom(Cluster_ Frequency) → Min_Volt...Max_Volt is the voltage supply for the region. The set {x | ∃a. a ∈ ran(mapping) ∧ x=dom(mapping ▷ {a})} represents the regions where applications are mapped. In fact, the set {x | ∃a. a ∈ ran(mapping) ∧ x=dom(mapping ▷ {a})} is of type ℙ1(NoC), which allows us to specify regions (clusters) instead of individual cells and to model the cluster agents in a simpler manner. For the complete events and invariants, the reader is referred to Appendix B.

6.2. The cluster agents: generic task reallocation within a cluster

The reallocation procedure performed by a cluster agent inside a region proceeds similarly to the reallocation procedure executed by the platform agent. That is, the cluster agent stores the trace between a faulty cell and its substitution and marks a faulty cell when its task has been reallocated. To specify this functionality, we introduce two variables. The first one keeps track of the faulty cells and their substitutions (one cell can have one substitution):

$$\textbf{invariant}\ \text{Cluster\_Cell\_Trace} \in \text{dom(mapping)} \rightarrowtail\mkern-18mu\twoheadrightarrow \text{dom(mapping)}$$

The second variable indicates that the task of a cell has been reallocated. The cluster agent adds to this variable the coordinates of a cell whose task has been migrated:

$$\textbf{invariant}\ \text{Cluster\_Cell\_Reallocated} \subseteq \text{dom(mapping)}$$

There is a clear relationship between these variables. Whenever the cell has a trace, this cell is marked and vice versa. This property is postulated as the invariant below:

$$\begin{aligned}\textbf{invariant}\ \forall c.\ c \in \text{dom(mapping)} \Rightarrow (&c\\ &\in \text{dom(Cluster\_Cell\_Trace)} \Longleftrightarrow c\\ &\in \text{Cluster\_Cell\_Reallocated})\end{aligned}$$

The task reallocation must be performed inside the region where the application is mapped. The cluster agent only utilizes the unallocated cores that belong to its region as it does not know anything about other applications and their mapping, i.e., the cross-cluster communication is not allowed:

$$\begin{aligned}\textbf{invariant}\ \forall a, \text{cell}.\ a \in \text{ran(mapping)} \wedge{} &\text{cell}\\ &\in \text{dom}(\text{mapping} \rhd \{a\}) \Rightarrow (\text{cell}\\ &\in \text{dom(Cluster\_Cell\_Trace)}\\ &\Rightarrow \text{Cluster\_Cell\_Trace(cell)}\\ &\in \text{dom}(\text{mapping} \rhd \{a\}))\end{aligned}$$

Clearly, any task migration procedure affects the allocation of tasks. Firstly, the cell that has a trace is not running computations, i.e., no task is assigned to it:

$$\begin{aligned}\textbf{invariant}\ \forall c.\ c \in \text{dom(Cluster\_Cell\_Trace)} \Rightarrow \neg &c\\ &\in \text{dom(app\_tasks\_allocation)}\end{aligned}$$

Secondly, the substituting cell must run computations. This is the primary function of a spare cell while the faulty cell is being reconfigured:

$$\begin{aligned}\textbf{invariant}\ \forall c.\ c \in \text{ran(Cluster\_Cell\_Trace)} \wedge \neg &c\\ &\in \text{Cluster\_Cell\_Reallocated} \Rightarrow c\\ &\in \text{dom(app\_tasks\_allocation)}\end{aligned}$$

Finally, the local reconfiguration procedure initiated by a corresponding cell agent commences when the task of a faulty core has been reallocated:

$$\begin{aligned}\textbf{invariant}\ \forall c.\ c \in \text{dom(mapping)} \Rightarrow (&\text{Cell\_Start\_Reconfig}(c)\\ &= \text{TRUE} \Rightarrow c \in \text{Cluster\_Cell\_Reallocated})\end{aligned}$$

At this refinement step, we introduce two events that model generic task reallocation within the cluster. Note that at this point, the cluster agent can reallocate a task to any unallocated core, which may not be at the rightmost column. The actual reallocation algorithm and its properties described in Section 4.3 are modeled in the next refinement.

The cluster agent reallocates a task in a non-deterministic manner upon the detection of a fault. The event Cluster_cell_reallocation specifies task reallocation from a faulty cell to a spare one:

```
event Cluster_cell_reallocation refines Restore_cluster ≙ any
   app x y k l
where
  // A faulty cell and its substitution are within the cluster
  app ∈ dom(app_ctallocated) ∧ k↦l∈dom(mapping ▷ {app}) ∧
   x↦y∈dom(mapping ▷ {app}) ∧
  // A faulty cell is running a task while a substitution does not
   run any
  k↦l∈dom(app_tasks_allocation) ∧
   ¬(x↦y∈dom(app_tasks_allocation)) ∧
  // A substituting cell is free
  (x↦y ∈ dom(Cluster_Cell_Trace)) ∧ ¬(x↦y ∈
   ran(Cluster_Cell_Trace)) ∧
  // The application of DVFS did not help
  Cluster_Frequency(dom(mapping ▷ {app})) = 0 ∧
   Cluster_Voltage(dom(mapping ▷ {app})) = Min_Volt ∧
  // Hence, the cell running a task is faulty
  (Cell_Temp)(k↦l) ⩾ Temp_Threshold ∨ Cell_Fault(k↦l) =
   (TRUE) ∧
  // while the substituting cell is not
  Cell_Temp(x↦y) < Temp_Threshold ∧ Cell_Fault(x↦y) =
   FALSE
then ... ||
  // Rellocate a task
  app_tasks_allocation := {k↦l} ⩤ (app_tasks_allocation ∪
   {x↦y↦app_tasks_allocation(k↦l)}) ||
  // Store a track from where to where the task has been
   reallocated
  Cluster_Cell_Trace := Cluster_Cell_Trace ∪ {(k↦l)↦(x↦y)} ||
  // Mark the faulty cell
 Cluster_Cell_Reallocated := Cluster_Cell_Reallocated ∪ {k↦l} ||
  // If the faulty cell happens to be the central one, update the
   corresponding mapping as well
  app_ctallocated:| app_ctallocated′∈ran(mapping)
  ⤖dom(mapping) ∧
     (¬app_ctallocated(app) = k↦l ⇒ app_ctallocated′ =
     app_ctallocated) ∧
     (app_ctallocated(app) = k↦l ⇒ app_ctallocated′ =
     app_ctallocated  {app↦(x↦y)})
end
```

We can observe from the event above that task migration occurs if the frequency and the voltage the cluster operates at are at their minimum values and a fault remains. Furthermore, the cluster agent reallocates a task from a faulty core to a spare one even if a fault occurs in the central core. This is because the reallocation takes place within the cluster so that the communication efficiency of the application remains at an adequate level.

The other event, namely Cluster_cell_return, models reallocation of a task back to the reconfigured cell according to the stored trace:

```
event Cluster_cell_return refines Restore_cluster ≙ any app x y k l
where ... ∧ // A reconfigured cell and its substitution are within the cluster
  // There is a track between a reconfigured cell and its substitution
  ¬(k↦l∈Cluster_Cell_Reallocated) ∧ ((x↦y)↦(k↦l))∈Cluster_Cell_Trace ∧
  // The cluster runs normally
  Cluster_Frequency(dom(mapping ▷ {app})) > 0 ∧ Cluster_Voltage(dom(mapping ▷ {app})) > Min_Volt ∧
  // A cell whose task was reallocated has been reconfigured and it is not faulty anymore
  Cell_Start_Reconfig(x↦y) = FALSE ∧ Cell_Temp(x↦y) < Temp_Threshold ∧ Cell_Fault(x↦y) = FALSE ∧
  // If there is only one substitution left, the reallocation back proceeds such that the rightmost column is released
   {(dom(mapping ▷ {app}) ∩ ran(Cluster_Cell_Trace ⩤ {k↦l}) = ∅ ⇒
     x↦y∈dom(dom(mapping▷{app}))×min(ran(dom(mapping▷{app})))..max(ran(dom(mapping▷{app})))−1)
then ... || // The variables app_tasks_allocation and app_ctallocated are modified exactly as in the event above
  // Remove the trace and unmark the reconfigured cell
  Cluster_Cell_Trace:={x↦y} ⩤ Cluster_Cell_Trace || Cluster_Cell_Reallocated:=Cluster_Cell_Reallocated\{x↦y}
end
```

6.3. The cluster agents: task reallocation within clusters based on task graphs

In the previous sub-section, we have introduced a generic reallocation functionality of the cluster agents. Now, we refine it so that the reallocation within the cluster proceeds in a more specific manner following the algorithm presented in Section 4.3. In particular, the cluster agent utilizes the spare cores available at the rightmost column of the region when migrating a task from a faulty cell. This crucial property is postulated as the invariant shown below:

$$\textbf{invariant}\ \forall a.\ a \in \mathrm{ran}(\mathrm{mapping}) \wedge (\mathrm{dom}(\mathrm{mapping} \rhd \{a\}) = \mathrm{dom}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})) \times \mathrm{ran}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})) \Rightarrow (\forall c.\ c \in \mathrm{dom}(\mathrm{mapping} \rhd \{a\}) \wedge c \in \mathrm{ran}(\mathrm{Cluster\_Cell\_Trace}) \Rightarrow \mathrm{ran}(\{c\}) = \{\max(\mathrm{ran}(\mathrm{dom}(\mathrm{mapping} \rhd \{a\})))\})$$

where (dom(mapping ▷ {a}) = dom(dom(mapping ▷ {a}))×ran(dom(mapping ▷ {a}))) shows that the shape of the cluster is consistent, ran({c}) stands for the column coordinate of the core c and {max(ran(dom(mapping ▷ {a})))} is the rightmost column of the region.

Following the algorithm described in Section 4.3, we refine the abstract event Cluster_cell_reallocation into several events that model different cases of task migration within the region. These events are similar and only differ in several guards that determine the manner the task is reassigned. Here, we show the textual description of guards and actions of some events. The complete subset of events modeling the reallocation algorithm within the cluster can be found in Appendix B.

The cluster agent initiates the task migration procedure when the decrease of frequency and/or voltage is not sufficient. The following event models all possible cases for the task reallocation of a top-most cell, i.e., the reallocation to a core allocated on a row below, to the core on the same row as the faulty one or to any available free spare core (see Fig. 8, a–c)):

```
event Cluster_top_cell_reallocation refines
  Cluster_cell_reallocation ≙ any app x l spares
where ... ∧
  // The top-most core is faulty
  // The core to be used as a substitution must not have a
   trace nor be a substition already
  // The regional parameters have reached their minimums
  // The cell from where the cluster agent reallocates the task
   is faulty
  // The cell where the task is reassigned is not faulty
  // The cluster agent utilizes spare cores from the rightmost
   column
  // If there is a cell below the current and this cell is free,
   reallocate the task there
  ((min(dom(dom(mapping ▷
   {app})))+1)∈dom(dom(mapping ▷ {app})) ∧
  (¬(min(dom(dom(mapping ▷ {app}))) +
   1)∈dom(ran(Cluster_Cell_Trace))) ⇒ x =
   min(dom(dom(mapping ▷ {app})))+1) ∧
  // If there is no such a cell or this cell is allocated move the
   task to the cell on the same row
  ((¬(min(dom(dom(mapping ▷
   {app})))+1)∈dom(dom(mapping ▷ {app})) ∨
    ((min(dom(dom(mapping ▷
   {app})))+1)∈dom(ran(Cluster_Cell_Trace)))) ∧
    ¬(min(dom(dom(mapping ▷
   {app})))∈dom(ran(Cluster_Cell_Trace))) ⇒ x =
   min(dom(dom(mapping ▷ {app})))) ∧
  // Finally, if none of these conditions are true, but there is a
   free spare, reallocate the task there
  ((¬(min(dom(dom(mapping ▷
   {app})))+1)∈dom(dom(mapping ▷ {app})) ∨
    ((min(dom(dom(mapping ▷
   {app})))+1)∈dom(ran(Cluster_Cell_Trace)))) ∧
    (min(dom(dom(mapping ▷
   {app})))∈dom(ran(Cluster_Cell_Trace))) ∧¬(spares = ∅) ⇒
   x∈spares)
then// Store the trace and Mark the faulty cell
  // Move the task to a spare core modifying the position of
   the central task, if needed
end
```

Similarly, the cluster agent reassigns a task from a faulty core allocated at the bottom of the region to a spare one in the rightmost column. The three options to reallocate a task are: to a core on a row above, to a core on the same row or to a core any available spare core. This case is modeled with a separate event, namely Cluster_bottom_cell_reallocation.

Finally, if a fault occurs in a cell that is neither top-most nor bottom-most, the cluster agent considers several cases of an efficient task migration. These cases are specified using several events. For the sake of brevity, we show only the essential ones. The others have the same structure and differ only in several guards.

As the first attempt, the cluster agent moves a task of a faulty core to a spare core, which is on the same row, but in the rightmost column. This core is equidistant from cores running computations (see Fig. 8d)):

```
event Cluster_middle_cell_reallocation_row refines
  Cluster_cell_reallocation ≙ any app k l
where . . . ∧
  // The faulty core is in the middle of the region
  min(dom(dom(mapping ▷ {app}))) < k ∧
  k < max(dom(dom(mapping ▷ {app}))) ∧
  // There is a free spare cell on the same row
  ¬(k ∈ dom(dom(Cluster_Cell_Trace))) ∧
  // Move the task to this core
then   // Store the track and mark the faulty cell
  // Move the task to a spare core modifying the position of the
  central task, if needed
end
```

However, if the spare cell on the same row is already utilized, the cluster agent considers several possibilities of task reallocation (see Fig. 8e)). In particular, the task can be reallocated to a spare cell on a row above or below depending on the number of links and/or communication bandwidth between a task allocated on a faulty core and adjacent cells.

Every task in a task graph has at most two transitions with another task. That is, it may have no transitions, one incoming or outcoming transition or both an incoming and an outcoming transition. Hence, it is reasonable to first consider if there are any transitions between a faulty core and the cores above and below it. If the number of transitions between a faulty core and the core below is greater than the number of transitions between a faulty core and the core above it, the cluster agent reallocates the task to the spare core on a row below. Consequently, there are only two hops between the spare core and the core with which there are two communication links:

```
event Cluster_cell_reallocation_spare refines Cluster_cell_reallocation ≙ any app x k l
where . . . ∧
  // If the spares on the current row, the row above and the row below are already allocated
  (∀e. e∈k−1..k+1 ⇒ e∈dom(dom(Cluster_Cell_Trace))) ∨
  // or there is a task assigned to a core above, but current task does not have transitions to it
  (((k−1)↦l∈dom(app_tasks_allocation) ⇒
  ¬(app_tasks_allocation(k−1↦l)↦app_tasks_allocation(k↦l) ∈ dom(app_trans(app))) ∧
    (app_tasks_allocation(k↦l)↦app_tasks_allocation(k−1↦l) ∈ dom(app_trans(app)))) ∧
  // nor with the core on the row below
   ((k+1)↦l∈dom(app_tasks_allocation) ⇒
    ¬(app_tasks_allocation(k+1↦l)↦app_tasks_allocation(k↦l) ∈ dom(app_trans(app))) ∧
    ¬(app_tasks_allocation(k↦l)↦app_tasks_allocation(k+1↦l) ∈ dom(app_trans(app)))))
  // Reallocate the task to any free spare core
  x ∈ {s|s∈min(dom(dom(mapping▷{app})))..max(dom(dom(mapping▷{app})))∧(s∈dom(ran(Cluster_Cell_Trace)))}
then // Store the trace and Mark the faulty cell
  // Move the task to a spare core modifying the position of the central task, if needed
end
```

Analogously, we have considered the other cases when there is only one transition with the upper and lower tasks. In total, there are four cases for a task: (i) two incoming transitions, (ii) two outcoming transitions, (iii) one incoming and one outcoming transition and (iv) one outcoming and one incoming transition (Fig. 11).

Finally, if none of these cases holds, the cluster agent simply reassigns the task from a faulty core to any available spare core:

```
event Cluster_middle_cell_reallocation_down1 refines
Cluster_cell_reallocation Restore_cluster ≙ any app k l
where . . . ∧
  // The spare cell on the same row is already assigned, but the spare cell on
a row below is free
  // There are tasks assigned to cores above and below the faulty one, i.e.
  // the cluster agent can try to verify if there are any links with those tasks
  // Acquire the transitions of the application task graph
  // Verify if the number of links with the task allocated below is greater than
the number of links with
  // task allocated above the task being reallocated
  // If so, move the task to the spare core on the row below
then   // Store the trace and Mark the faulty cell
  // Move the task to a spare core modifying the position of the central task,
if needed
end
```

After a task has been reallocated, a corresponding cell agent initiates the local reconfiguration procedure. When this procedure is complete and the cell is reconfigured, the cluster agent returns the task in accordance with the stored trace. Hence, the original communication performance within the region is restored:

```
event Cluster_cell_return refines Cluster_cell_return ≙ any app
  x y k
where . . . ∧
  // The substituting cell has a task
  ¬(k↦max(ran(dom(mapping ▷
  {app})))∈Cluster_Cell_Reallocated)
  // Return a task according to the stored track
  ((x↦y)↦(k↦max(ran(dom(mapping ▷
  {app}))))))∈Cluster_Cell_Trace ∧
then // Move the task back according to the stored track,
  remove the track and unmark the reconfigured cell
end
```

While modeling the cluster agents within Event-B, the Rodin platform [7] has generated 708 proof obligations of which 537 were

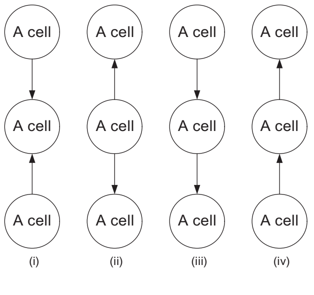


Fig. 11. Variations of transitions between two communicating tasks.

proven automatically (more than 75%). Table 5 summarizes the proof statistics. Similarly to the platform agent, the proof obligations (INV) and (FIS) (Section 3.1) for invariant preservation and feasibility required user assistance in showing correctness of the cluster agents specification.

Let us now examine the implementation of the cell agent that performs the local reconfiguration procedure.

7. Formal development of the cell agents

7.1. The cell agents: the base model

For the work in this paper, we adopt and adjust the specification of the cell agents proposed in [9] such that hardware code can eventually be generated as motivated in Section 4.4. This code can then be synthesized using, for instance, Quartus-II software [33]. Previously, we have used total functions to represent all the cell agents of the NoC platform. Because of that, the functionality of the cell agents was modeled using non-determinism on events (events with local parameters). Since the cell agents are the same independently of their location in the platform, we can proceed with further refinements towards a deterministic and implementable specification of a particular cell agent in order to generate a VHDL description [26] for a cell agent. To achive this, we first provide specific coordinates of the agent. Then, we eliminate functions so that code generation is feasible. Here, we only show the essential parts for deriving an implementable model from which a VHDL description can be generated. For the more detailed formal description, the reader is referred to Appendix C.

7.2. The cell agents: specifying coordinates of a cell agent

To specify coordinates of a cell agent, we introduce two constants in a context. Note that we do not specify the actual location due to the fact that the cell agents are the same. Hence, the location is not important:

constants n m
axioms $n \in 1..IPnum \wedge m \in 1..IPnum$

Then, we eliminate non-determinism of the **any** clause of every event modeling the functionality of a cell agent. For instance, consider the event modeling the beginning of the local reconfiguration. It operates only on the cell whose coordinates are the specified constants n and m. In other words, the event is deterministic and executes at the same coordinates in the platform:

Table 5
The cluster agents: proof statistics.

Model	Number of proof obligations	Automatically discharged	Interactively discharged
Contexts	9	7	2
The base machine	144	99	45
The first refinement	143	91	52
The second refinement	412	340	72
Total	708	537	171

event Reconfigure_cell **refines** Reconfigure_cell $\triangleq$
where
  $Cell\_Read(n \mapsto m) = TRUE \wedge (Cell\_Fault(n \mapsto m) = TRUE \vee$
  $Cell\_Temp(n \mapsto m) \geqslant Temp\_Threshold) \wedge$
  $Reallocated(n \mapsto m) = TRUE \wedge Reconfigured(n \mapsto m) = FALSE$
then $Start\_Reconfig := Start\_Reconfig \mathbin{<\!\!+} \{(n \mapsto m) \mapsto TRUE\}$
end

Now, we can refine this specification further by substituting functions with simply typed variables. These variables reflect the inputs and the outputs of the cell agent while their types are feasible for code generation.

7.3. The cell agent: substituting functions with simply typed variables and code generation

The cell agent can be represented as a block diagram as shown in Fig. 12. Since the inputs cannot be updated and the outputs cannot be read directly in the hardware code, we propose to use external loopback connections on the variables (signals) Start_Reconfig and Cell_Read (see Fig. 12).

From Fig. 12 we observe the inputs and the outputs the cell agents have. The variables with the suffix "_I" are the input signals while the variables with suffix "_O" indicate the output signals. The inputs and the outputs are represented as simply typed variables, where most variables are of Boolean type. The only variable that has the numeric type is "Temp_I" as it models the changes on the temperature of the cell: $Cell\_Temp\_I \in 0..Temp\_max$. The variables $Cell\_Fault\_I \in BOOL$, $Reallocated\_I \in BOOL$ stand for the faults that occur in the cell and the reallocation of the task performed by the higher level agents, respectively. To model a loopback, we introduce the following variables:

invariant $Cell\_Read\_I \in BOOL \wedge Cell\_Read\_O \in BOOL \wedge$
  $Start\_Reconfig\_I \in BOOL \wedge Start\_Reconfig\_O \in BOOL$

Since the decomposed model also includes external events that simulate the environment for the cell agents, we have to refine these events accordingly. To proceed correctly, we introduce functions that are marked as external so that the code generation tool skips these variables when deriving VHDL. For instance, the variable Cell_Temp modeling the temperature of a cell is refined (replaced) by the variable Cells_Temp.

To prove the correctness of the refinement and derive the implementable model used to generate VHDL code, we introduce several gluing invariants. These invariants state that the value of every simply typed variable conforms to the value of a function applied to the cell for which we have defined the constants. On the

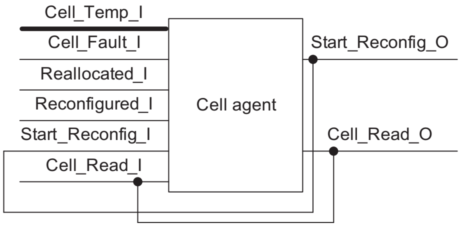


Fig. 12. The hardware representation of a cell agent.

```
event Reconfigure_cell refines Reconfigure_cell ≙
where
   Cell_Read(n↦m) = TRUE ∧
   (Cell_Fault(n↦m) = TRUE ∨
   Cell_Temp(n↦m) ≥ Temp_Threshold) ∧
   Reallocated(n↦m) = TRUE ∧
   Reconfigured(n↦m) = FALSE
then
   Start_Reconfig ≔ Start_Reconfig ⩤- {(n↦m)↦TRUE}
end
```

```
event Reconfigure_cell refines Reconfigure_cell ≙
where
   Cell_Read_I = TRUE ∧
   (Cell_Fault_I = TRUE ∨
   Cell_Temp_I ≥ Temp_Threshold) ∧
   Reallocated_I = TRUE ∧
   Reconfigured_I = FALSE
then
   Start_Reconfig_O ≔ TRUE || Start_Reconfig_I ≔ TRUE
end
```

Fig. 13. The event Reconfigure_cell.

other hand, the abstract functions of the previous model remain consistent:

invariant Cell_Temp = ({n↦m} ⩤ Cells_Temp) ∪ {n↦m↦Cell_Temp_I}

The gluing invariant states that the value of the abstract function Cell_Temp is the same as the union of the values of the concrete function Cells_Temp without the coordinates n, m and the variable Temp_I that complements the former. This approach allows us to isolate a particular cell from the set of all the cells in the platform.

Analogously, we refine the other variables except for the variables that model a loopback. For the variable Start_Reconfig, we provide a similar invariant to the one shown above as well as we postulate that the value of the output is the same as the input:

invariant Start_Reconfig = ({n↦m} ⩤ Cells_Start_Reconfig) ∪
{n↦m↦Start_Reconfig_I} ∧
Start_Reconfig_O=Start_Reconfig(n↦m)

Since the variable Cell_Read modeling asynchronous communication between the agents in the hierarchy is not affected by the event simulating the environment at the coordinates different from the ones specified by the constants, the gluing invariants for it are simpler:

invariant Cell_Read_I = Cell_Read(n↦m) ∧ Cell_Read_O =
Cell_Read(n↦m)

Consequently, in every event we have replaced a function call with the precise variable. For instance, consider the event Reconfigure_cell (Fig. 13).

According to the gluing invariants, this event behaves exactly the same as its ancestor. Notice, however, that this event has a simultaneous assignment to both output and input variables Start_Reconfig_I and Start_Reconfig_O. This action models the loopback over the reconfiguration command (see Fig. 13).

Although such an action allows us to model a loopback connection, the actual implementation in VHDL cannot have assignments to the input signals. Hence, when generating the code, the tool [26] skips the assignments to the input variables and keeps the updates on the output variables. Additionally, the tool omits external variables and events as they are not relevant to the cell agent functionality. The example of the generated code for the event Reconfigure_cell is shown below:

```
IF (Cell_Read_I = '1') and
   (Cell_Fault_I = '1' or Cell_Temp_I > = Temp_Threshold)
 and
   (Reallocated_I = '1') and
   (Reconfigured_I = '0')
THEN
   Start_Reconfig_O <= '1';
END IF;
```

In these events and the generated code, the cell agent has one input displaying that a task has been reallocated. However, one can proceed with an additional refinement step where this single input is split into two: one coming from the platform agent and the other one coming from a corresponding cell agent. Since this refinement is easily performed through data refinement, we omit it.

The proof statistics for this model is summarized in Table 6. From the table, we observe that the Rodin platform [8] has generated 291 and has discharged 275 of them automatically. The proof statistics illustrate that the tool could achieve a high level of automated proving (more than 95%). The interactive proofs include discharging proof obligations feasibility, guard strengthening and simulation (Section 3.1, POs (FIS), (GRD) and (SIM), respectively) due to substituting data structures.

8. Conclusions

In this paper, we have presented the formal modeling and verification of a hierarchical agent-based dynamic management system for NoCs incorporating application mapping and a novel task reallocation procedure utilizing free spare cores available to each running application. Specifically, the hierarchical agent-based

Table 6
The cell agent: proof statistics.

Model	Number of proof obligations	Automatically discharged	Interactively discharged
The contexts	4	3	1
The base machine	89	87	2
The first refinement	83	82	1
The second refinement	115	106	9
Total	291	278	12

management system consists of the platform agent, a number of dynamically created and destroyed cluster agents and local cell agents. We proposed (i) for the platform agent level, an algorithm for the initial application mapping and tasks allocation with free spare cores, (ii) for the cluster agent level, a multi-objective algorithm that facilitates fault-tolerance of the platform while maintaining performance of communication and computations at an adequate level, (iii) for the cell agent level, an algorithm integrating the local reconfiguration procedure for a cell. The distributed architecture of the cell and cluster agents allows independent execution of monitoring and reconfiguration procedures utilizing the spare cores without overloading the platform agent.

The development of each agent level proceeded through refinements considering the overall requirements for the system. To the best of our knowledge, this is the first approach for developing reliable agent-based management systems for dynamically reconfigurable NoC platforms that incorporates a formal and proof-based framework. The important functional properties (requirements) of the system have been stated as invariants and the corresponding proof obligations have been discharged. Therefore, we have verified the correctness of the proposed system w.r.t. these stated properties.

We considered mapping of an application in such a manner that a rightmost column of spare cores is at the disposal of a corresponding cluster agent. One of our future directions is to investigate other possibilities of placing spare cores within the region. For instance, spare cores can be placed randomly following the approach presented in [12]. In addition, we will exploit reallocation of tasks to unallocated spare cores within other clusters.

We have derived a VHDL description for the cell agents following a correct-by-construction development. Similarly, we can eventually derive implementations for the other agents. Hence, another future direction is to evaluate efficiency of the proposed system considering specific metrics.

Last but not least, a fault can also occur in the agents, although the agents are much simpler than the processing cores. Therefore, another future direction is to consider faults of agents and their dynamic reconfiguration.

Acknowledgments

The authors would like to thank Linas Laibinis for valuable feedback on the formal development. The authors would also like to thank the reviewers for constructive comments. The work is partially supported by Academy of Finland and Research Institute of Åbo Akademi University.

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.sysarc.2013.06.001.

References

[1] L. Zhang, Y. Han, Q. Xu, X. wei Li, H. Li, On topology reconfiguration for defect-tolerant NoC-based homogeneous manycore systems, IEEE Transactions on Very Large Scale Intergrations (VLSI) Systems 17 (9) (2009) 1173–1186.

[2] I. Khatib, D. Bertozzi, F. Poletti, L. Benini, A. Jantsch, M. Bechara, H. Khalifeh, M. Hajjar, R. Nabiev, S. Jonsson, MPSoC ECG biochip: a multiprocessor system-on-chip for real-time human heart monitoring and analysis, Conference on Computing Frontiers, ACM, New York, 2006, pp. 21–28.

[3] K. Motamedi, N. Ionnides, M. Rümmeli, I. Schagaev, Reconfigurable network on chip architecture for aerospace applications, in: Preprints of the 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software, 2009, pp. 131–136.

[4] P. Rantala, J. Isoaho, H. Tenhunen, Novel agent-based management for fault-tolerance in network-on-chip, Euromicro Conference on Digital System Design Architectures, Methods and Tools, IEEE, Lubeck, 2007, pp. 551–555.

[5] A. Yin, L. Guang, P. Liljeberg, E. Nigussie, J. Isoaho, H. Tenhunen, Hierarchical agent based NoC with dynamic online services, Industrial Electronics and Applications ICIEA, IEEE, Xi'an, 2009, pp. 434–439.

[6] Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, IEC61508, 2010.

[7] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, Cambridge, 2010.

[8] RODIN, July 18, 2012. Available: <http://sourceforge.net/projects/rodin-b-sharp/>.

[9] S. Ostroumov, L. Tsiopoulos, Formal development of hierarchical agent-based monitoring systems for dynamically reconfigurable NoC platforms, International Journal of Embedded and Real-Time Communication Systems (IJERTCS) 1 (2) (2012) 40–72 (IGI).

[10] L. Guang, J. Plosila, J. Isoaho, H. Tenhunen, Hierarchical agent monitored parallel on-chip system: a novel design paradigm and its formal specification, International Journal of Embedded and Real-Time Communication Systems (IJERTCS) 1 (2) (2010) 86–105 (IGI).

[11] L. Guang, S. Jafri, B. Yang, J. Plosila, H. Tenhunen, Embedding fault-tolerance with dual-level agents in many-core systems, in: MEDIAN Workshop, EU COST Action IC1103 Median, 2012, pp. 41–44.

[12] C.-L. Chou, R. Marculescu, FARM: fault-aware resource management in NoC-based multiprocessor platforms, Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, Grenoble, 2011, pp. 1–6.

[13] P. Hölzenspies, T. Braak, J. Kuper, G. Smit, J. Hurink, Run-time spatial mapping of streaming applications to heterogenous multi-processor systems, International Journal on Parallel Programming (2009) 68–83.

[14] S. Le Beux, G. Bois, G. Nicolescu, Y. Bouchebaba, M. Langevin, P. Paulin, Combining mapping and partitioning exploration for NoC-based embedded systems, in: Journal of Systems Architecture (JSA), Elsevier, New York, 2010, pp. 223–232.

[15] C. Métayer, J.-R. Abrial, L. Voisin, Deliverables. Rigorous Open Development Environment for Complex Systems, May 31, 2005. Available: <http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf>.

[16] K. Robinson, System Modelling & Designing using Event-B, June 28, 2011. Available: <http://www.cse.unsw.edu.au/~cs9116/PDF/SMD.pdf>.

[17] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, Computer: IEEE 35 (1) (2002) 70–78.

[18] C.-L. Chou, R. Marculescu, User-aware dynamic task allocation in networks-on-chip, Design, Automation and Test in Europe DATE, IEEE, Munich, 2008, pp. 1232–1237.

[19] S. Shamshiri, P. Lisherness, S.-J. Pan, K.-T. Cheng, A cost analysis framework for multi-core systems with Spares, Test Conference ITC, IEEE, Santa Clara, 2008, pp. 1–8.

[20] Altera. FPGA Architecture, July 2006. Available: <www.altera.com/literature/wp/wp-01003.pdf>.

[21] R. Hartenstein, Coarse grain reconfigurable architectures, Asia and South Pacific Design Automation Conference, ACM, New York, 2001, pp. 564–570.

[22] B. Yang, T. Xu, T. Säntti, J. Plosila, Tree-model based mapping for energy-efficient and low-latency network-on-chip, Design and Diagnostics of Electronic Circuits and Systems (DDECS), IEEE, Vienna, 2010, pp. 189–192.

[23] C. Pascal, R. Silva, Event-B Model Decomposition, November 30, 2009. Available: <http://eprints.soton.ac.uk/69664/>.

[24] S. Hallerstede, T.S Hoang, Refinement by interface instantiation, International Conference on Abstract State Machines, ALLOY, B, VDM and Z (ABZ), LNCS, vol. 7316, Springer-Verlag, 2012, pp. 223–237.

[25] L. Guang, E. Nigussie, P. Rantala, J. Isoaho, H. Tenhunen, Hierarchical agent monitoring design approach towards self-aware parallel systems-on-chip, ACM Transactions on Embedded Computing Systems (TECS), vol. 9 (Issue 3), ACM, New York, 2010, p. 24.

[26] S. Ostroumov, L. Tsiopoulos, VHDL code generation from formal Event-B models, in: International Conference on Digital System Design (DSD), 14th Euromicro Conference, IEEE, Oulu, 2011, pp. 127–134.

[27] M. Palesi, D. Patti, F. Fazzino, Noxim, April 24, 2012. Available: <http://noxim.sourceforge.net/>.

[28] Xilinx, Remote FPGA Reconfiguration Using MicroBlaze or PowerPC Processors, September 9, 2006. Available: <http://www.xilinx.com/support/documentation/application_notes/xapp441.pdf>.

[29] Altera, Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs, July 2010. Available: <http://www.altera.com/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf>.

[30] K. Robinson, A concise summary of the Event-B mathematical toolkit, October 7, 2010. Available: <http://wiki.event-b.org/images/EventB-Summary.pdf>.

[31] R. Hilbrich, J. Reinier van Kampenhout, Dynamic reconfiguration in NoC-based MPSoCs in the avionics domain, in: Proceedings of the 3rd International Workshop on Multicore Software Engineering, ACM, New York, 2010, pp. 56–57.

[32] R. Hilbrich, J. Reinier van Kampenhout, Partitioning and task transfer on noc-based many-core processors in the avionics domain, in: Entwicklung zuverlässiger Software-Systeme Workshop, Stuttgart, Germany, 2011.

[33] Quartus-II software. Available: <http://www.altera.com/products/software/sfw-index.jsp>.

Paper 3

Derivation of Parallel and Resilient Programs from Simulink Models

Sergey Ostroumov, Pontus Boström, Marina Waldén

Originally published in:
Proceedings of International Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE Computer Society Conference Publishing Services (CPS), pp. 416-420, 2015.

Extended version published in:
Sergey Ostroumov, Pontus Boström, Marina Waldén, Mikko Huova, Deriving Efficient and Dependable Parallel Programs from Simulink Models. TUCS Technical Reports 1111, TUCS, 2014.

© 2015 IEEE. Reprinted, with permission, from IEEE.

2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Derivation of Parallel and Resilient Programs from Simulink Models

Sergey Ostroumov[1,2], Pontus Boström[1], Marina Waldén[1]
[1]Department of Information Technologies
Åbo Akademi University
Turku, Finland
{Sergey.Ostroumov, Pontus.Bostrom, Marina.Walden}@abo.fi
[2]TUCS – Turku Centre for Computer Science

***Abstract*—Modern embedded applications often require high computational power and, on the other hand, fulfilment of real-time constraints and high level of resilience. Simulink is one widely used tool for model-based development of embedded software. In this paper, we focus on the derivation of parallel programs from Simulink models and real-time resilient execution of derived implementations on a many-core platform. The main contribution is a fault-tolerance (FT) mechanism that prevents data loss when the platform is dynamically reconfigured to mask failures of individual cores. Finally, we evaluate the proposed solutions on an industrial case study using a commercially available NoC-based platform. The evaluation shows that the proposed FT mechanism has a marginal overhead.**

Keywords—Data Loss Prevention; Dynamic Reconfiguration; Many-Core Platforms; Parallel Programs; Resilience; Simulink

I. INTRODUCTION

The highly dynamic nature of modern embedded applications requires high computational power while they also need to fulfil real-time constraints and a high level of resilience. To develop such systems, designers typically employ various modelling techniques. The Simulink model-based design environment [1] is one such widely used technique that supports a complete design chain starting from modelling and simulation and ending in generation of, e.g., C code. However, the programs generated by the built-in code generator cannot fully utilize computational power offered by energy-efficient many-core platforms.

A Network-On-Chip (NoC) which represents a communication network of cores has been proposed as a scalable paradigm that can provide high computational power and low power consumption [2]. For instance, a commercially available platform TilePro by Tilera [3] employs NoC. However, the high level of on-chip integration increases the probability of various faults [4] and high computational load may cause creation of hotspots leading to thermal problems [5]. Additionally, radiation which is frequent in space but becomes an issue at the ground level as well can cause transient faults [6]. This can eventually induce a faulty execution of applications. One of the powerful techniques to tolerate these faults is dynamic reconfiguration, namely tasks reallocation [4][7][8]. This technique can be executed by the agents that are integrated into the platform and perform efficient management without overloading the platform with monitoring and recovering activities [8][9]. However, when tasks are reallocated to non-faulty cores, they may lose data in the process, which can lead to the production of an erroneous output. Consequently, to achieve resilience, application tasks need to adopt a mechanism that provides means to continue execution without losing data when they are reallocated.

Our contributions in this paper are: 1) an approach to the derivation of parallel implementations from Simulink models, 2) based on 1), a fault-tolerance (FT) mechanism that prevents data loss when application tasks are dynamically reallocated. In addition, we illustrate performance evaluation results for the proposed approaches by using the TilePro platform [3].

II. RELATED WORK

A Simulink model is a hierarchical dataflow diagram from which the Simulink design environment can generate sequential or fixed-priority multi-task C code scheduled according to the rate monotonic principle [1]. However, the generated code is not aimed at the parallel execution on a many-core platform.

In contrast to [1], we propose to generate a parallel implementation from a Simulink model by using application characteristic graphs (ACG) [7] as an intermediate step. The use of ACG allows designers to employ mapping algorithms for many-core platforms considering various optimization objectives, e.g., performance (real-time constraints) and/or power consumption [10], resilience [4][8] etc. The generated concurrent code preserves the semantics of Simulink models. Moreover, the division of the system into parallel tasks enables the application of resilience mechanisms to tasks and, hence, improves the utilization of the platform.

To achieve resilience to faults, redundancy is needed. For example, Bolchini, Carminati and Miele [6] propose to replicate the whole application or some of its threads in order to detect and tolerate failures of processors. They assume data parallel programs and consider duplication with comparison, triplication, as well as duplication with both comparison and re-execution FT techniques. The authors propose the adaptation engine that monitors several parameters and adapts the execution according to the evolving environment.

Pinello, Carloni and Sangiovanni-Vincentelli proposed another approach to replicating dataflow actors [11]. The authors consider a fault model, in which components are fail-silent, i.e., they either produce a correct result or produce no result. The authors use software replication for critical tasks statically at design time, where each replica is then executed on a separate control unit.

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.102

An approach to tackle hardware failures in process networks has been proposed by Ceponis, Kazanavicius and Mikuckas [12]. The authors present an extension of Kahn process networks, namely Error-Proof Process Network (EPPN). Using EPPN, the authors show a dynamic reconfiguration mechanism, where the actions of a faulty node are transferred to an adjacent non-faulty functional node and communication is adjusted accordingly using checks on the FIFO channels. However, according to the authors, this mechanism may lead to non-determinism in the network. Moreover, when functionality of a failed node is delegated to a non-faulty operating node, data loss occurs. To tackle this problem, the authors introduce the default value. Although the mechanism fulfils on-time result delivery, the default value may not preserve semantics of the original application.

Similarly as in [6][11][12], we consider hardware failures of processing units in the underlying many-core NoC-based platform. However, in contrast to [6][11][12], we rely on dynamic tasks reallocation [4][8] that can be performed by agents integrated into the platform [8][9]. The tasks reallocation enables uninterruptable execution of applications [4][7][8] and avoids resource wasting caused by duplicating applications or threads (actors). To avoid data loss when tasks are reallocated, we propose an FT mechanism, in which the reallocated tasks operate on the current values instead of the default ones in contrast to [12]. Therefore, the determinism of the application is preserved.

III. Preliminaries

A. Simulink Models

We consider Simulink models that represent hierarchical dataflow diagrams [1]. A Simulink model consists of a collection of functional blocks that have in-ports (inputs) and out-ports (outputs) allowing connections between blocks via typed signals. The blocks may have parameters that are initialized at the beginning of the execution and remain constant during the execution. Moreover, the blocks can contain memory. In this case, the output value depends not only on the inputs but also on the previously computed value.

The blocks can be grouped into sub-systems. There are two types of sub-systems in Simulink: virtual and atomic [1]. Virtual sub-systems are used for the structural purpose only and do not affect the model execution. They can be seen as containers for functional blocks that are expanded by the Simulink engine in place before execution. Atomic sub-systems are treated as single atomic units.

The models can be continuous or discrete. We consider discrete-time models with atomic sub-systems that specify periodic real-time systems. Each block in a discrete-time model is evaluated at regular intervals with a specified sampling period. We further assume that the model is single-rate, i.e., all its sub-systems fire at the same time intervals. In addition, we assume causal models, where outputs of a block have no direct connection to inputs of the same block. The models usually used for code generation are causal.

Fig. 1 illustrates an example of a Simulink model. The model in Fig. 1, a) contains two in-ports and one out-port. It includes a constant parameter as well as a memory block. This model is grouped into a sub-system presented in Fig. 1, b).

B. Communication platform

The generation of a parallel code requires designers to take into account characteristics of the underlying platform. We assume a 2D mesh NoC-based many-core platform. It consists of tiles that include processing units (PUs) and routers (RTs) [2] (Fig. 2). We assume the platform to be homogenous at the global level, i.e., all tiles are identical, while their internal structure might be heterogeneous.

RTs allow communication between tiles by routing packets. We assume deterministic routing, which is dead-lock and live-lock free, provides low latency and suits real-time control systems [13]. The communication mechanism typically employs FIFO buffers, which preserves the flow order of data. Moreover, the platform typically supports checks if the buffers are full or empty. Thus, the tasks can read packets as soon as they arrive in the input buffers and send processed data when there is an available space.

IV. Derivation of Parallel Programs from Simulink Models

We translate a Simulink model into a set of concurrent tasks that are given by the sub-systems and communicate according to the signals in between. This process can be summarized as the following algorithmic steps:

1. Flatten the model following so that the top-level atomic sub-systems reflect tasks according to the designer choice.
2. Construct an ACG from the model as explained in the next sub-section.
3. Generate implementations (i.e., threads) for the tasks according to ACG.
4. Apply mapping algorithms using the ACG [4][8][10].

Here, we focus on steps 2 and 3.

A. Construction of ACG from Simulink

To apply mapping algorithms that enable optimization in terms of, e.g., performance and power consumption [10] or resilience [4][8], we need to construct an Application Characteristic Graph (ACG) from a flattened Simulink model. An ACG consists of tasks and edges. The edges show communication rates r between tasks via FIFOs. For brevity, we only provide an intuitive description of the ACG derivation from a Simulink model. Please refer to the technical report [18] for more details on the construction.

The construction of an ACG from an arbitrary model is illustrated in Fig. 2. Similarly as in the approach proposed by Boström [14], we interpret each node of the model as a vertex of ACG with synchronous dataflow semantics, i.e., each atomic sub-system as a separate execution task that can be run on a single core. However, in contrast to [14], we group the links of the Simulink model into edges of ACG. An edge between an arbitrary pair of nodes in ACG reflects a group of

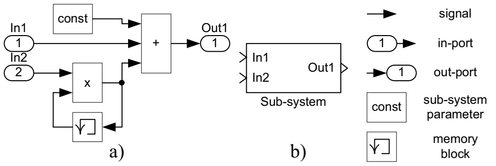


Fig. 1. Simulink models: a) sub-system content, b) sub-system block

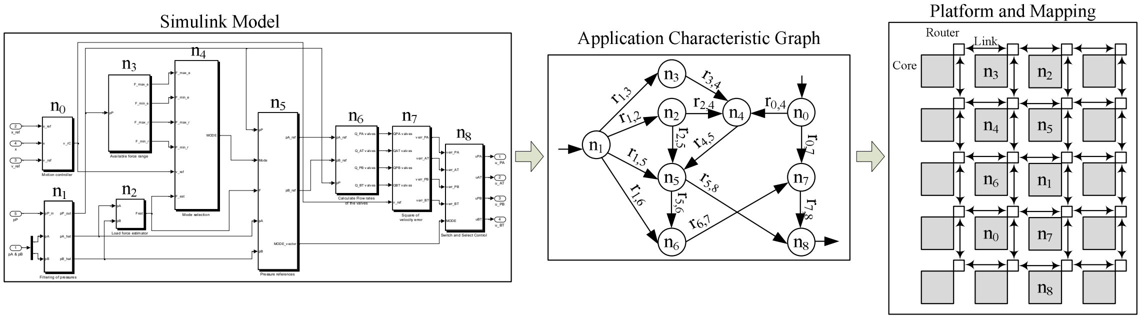


Fig. 2. Application characteristic graph and mapping example

links between the same nodes in the model. In essence, the links constitute communication between the nodes. The rates of packets are computed according to the execution periods of the corresponding sub-systems. The input and the output signals of the blocks that interact with the environment do not participate in the construction of ACG. This is because these signals do not affect the application internal structure.

B. *Task pattern*

Each task of the ACG executes a function and is mapped to a separate PU in the platform. However, despite different functionality, each task instantiates the pattern shown in Fig. 3. A task runs the loop for Receiving, Processing and Sending (RPS) data:

- a task starts processing data as soon as it has at least one token (i.e., one piece of data) in every input FIFO buffer,
- when a task runs, it consumes one token from every input buffer and produces one token for every output buffer, i.e., the task processes the received data according to the function derived from the model and sends processed data further according to the edge of ACG,
- a task without inputs fires every t sampling time.

To preserve timing semantics between a Simulink diagram and ACG, we assume that the computation and communication time in ACG equals to 0 as it does in the Simulink blocks and links.

V. Resilience of The Platform and Applications

To achieve resilience to faults and maintain performance, various dynamic reconfiguration techniques are utilized. One such powerful technique is tasks reallocation [4][7][8], where a task is migrated to a non-faulty PU when some PU fails. However, when tasks are reallocated, they may lose data.

Let us now focus on a resilience mechanism that prevents the data loss in the type of ACG considered in this paper. We start by describing faults and fault scenarios that can occur.

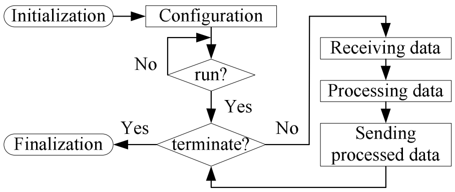


Fig. 3. Task pattern

A. *Fault model*

We consider the fault model that captures *physical failures of processing units* of the platform. A failure can be caused by transient, intermittent or permanent faults due to high temperature [5], radiation [6], etc. We assume that only one failure of PU can occur at a time independently of the number of faults causing it. In other words, a sufficient amount of time must elapse between two consecutive failures.

For the sake of simplicity, we assume that PUs are fail silent that either produce the correct result or no result at all [11][15]. Fail-silence assumption however can be softened if erroneous results are detected and isolated by using various mechanisms such as model-based diagnosis [16], runtime verification [17] or by integrating CRC-like sums into packets and their checks into tasks [15].

After a task is reallocated from a failed tile, the task starts over from the initialization phase (see Fig. 3); hence, all local variables receive initial values. However, the packets are stored in the buffers of RT which is a separate unit of a tile (see, e.g., [3]) or in the main memory. Therefore, these data remain intact and can also be reallocated along with the task.

We can assume that reading from and writing to a FIFO buffer (queue) are atomic operations, i.e., either the buffer is read or updated, respectively, or not. However, if a task has several input and/or several output buffers, the reading and sending proceed in a buffer-by-buffer manner. In addition, we distinguish between source and regular tasks. The source tasks receive input data from the environment. The regular tasks consume data produced by other tasks and send processed data further or provide an output to the environment. Independently of whether a task is source or regular, it can be stateless (without memory) or stateful (with memory). Consequently, we have 4 cases in total: *stateless regular* tasks, *stateless source* tasks, *stateful regular* tasks and *stateful source* tasks.

B. *Fault scenarios*

According to the described fault model, there are several possible *fault scenarios* (FS) for the fault occurrence within the RPS loop (see Fig. 3):

(FS1) *A fault occurs before a task reads any input data.* In this case, a task can still read the input data after reallocation as the input data remain intact.

(FS2) *A fault occurs while a task reads input data.* A task reads packets from some queues but fails to read from others. Thus, some pieces of data may be lost.

(FS3) *A fault occurs before the task sends the processed data.* The task has read all the input data but has not finished processing them or has not been able to send the processed data. Hence, the task loses data of one firing.

(FS4) *A fault occurs while a task sends data.* In this case, some successor tasks may receive packets with new data while others may not. This can lead to desynchronized data reception by the successor tasks.

C. Packet sending

To address data loss according to the described above FSs, we propose the following mechanism. Firstly, the packets used for communication between tasks incorporate a sequence number (packet id). The source tasks provide a value for this number starting from 0 and increase it every time when a new input is read. The regular tasks do not change this number which allows tasks to synchronize packets received from different buffers as explained later.

Secondly, every task except for the ones that produce the output to the environment sends the same packet twice: the main packet and its duplicate (Fig. 4). Please notice however that the same approach can also be applied if tasks send more duplicate packets to tolerate a larger number of faults of PUs.

As a result, the tasks now send and receive two packets instead of one each time they fire. The packets integrate a sequence number (id), where the main packet and its duplicate have the same id. However, the tasks need an intelligent procedure that filters duplicates if the tasks operate normally and use them upon failure according to the described FSs.

For brevity, we provide a detailed description of intelligent packet handling for the stateless regular tasks in the following sub-section. Please refer to the technical report [18] for the details on the other types of tasks.

D. Intelligent packet handling for stateless regular tasks

Stateless regular tasks operate according to the algorithm presented in Fig. 5. In order for a task to detect a duplicate packet, it stores a local copy (lsn) of packets id after receiving all input packets. The initial value of the local copy equals to -1 so that it is less than the starting value of the packets id (i.e., 0). When reading packets, the task compares the id of the packet just read with the local copy. If no fault has occurred, the value of the local copy of the sequence number is less or equal to the sequence number of the packets read. Since the id of the main packet and its duplicate is the same, the task will simply reread the buffer for a packet with a greater id (Fig. 5, condition $pkt\_q_j.id <= lsn$). This packet will contain new input data to process, i.e., be a new main packet. Thus, the task will filter duplicates when operating normally (see Fig. 6).

In case a fault occurs, the local copy of the sequence number is initialized with -1. Depending on the FS, there are several possible outcomes. In (FS1), the task proceeds normally after reallocation as the main packets remain intact in the input buffers (see Fig. 4). The effect of the other FSs is shown in Fig. 6 which captures states of the input buffers of task n_i considering (FS2)-(FS4).

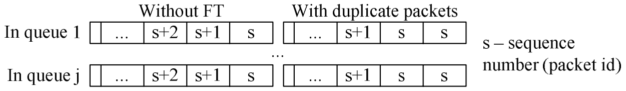


Fig. 4. State of buffers with and without FT

If (FS2) takes place, there are two possible cases. In the first one, a fault occurs while the task reads main packets from buffers (Fig. 6, FS2, Case 1). In this case, the task can proceed normally after reallocation since there are duplicates in the buffers. In the second case, a fault occurs when the task has read duplicate packets from some queues but failed to read duplicates from other queues (Fig. 6, FS2, Case 2). This may lead to desynchronized packet receiving as the task reads data in a buffer-by-buffer manner. To avoid this, the task compares packet id received from the first queue with ids of the packets read from other queues. If the id of a packet from another queue is less than the id of a packet from the first queue, the task needs to reread this queue (Fig. 5, condition $pkt\_q_j.id < pkt\_q_1.id$). This enables synchronization of packets read from different queues as only source tasks provide sequence numbers for packets and regular tasks do not modify them.

In (FS3), where a fault occurs before the task starts sending the processed data, the task will use duplicates residing in the buffers after reallocation (Fig. 6, FS3).

Finally, the algorithm also covers (FS4) if, e.g., task n_l is reallocated due to a failure of PU (Fig. 6, FS4), as at least one copy of a packet always resides in the buffers. Please notice that a task can send more than two duplicates in case of (FS4). However, they will be filtered by the proposed algorithm.

VI. Evaluation Results

The proposed approach has been evaluated on a case study [18] implemented on the TilePro platform [3] without running other applications than OS (Linux Santiago 6.0, Kernel 2.6.36-4). The platform integrates 64 tiles forming an 8x8 square mesh with a network-based communication between the tiles. The network connections are 32-bit full-duplex, there is single cycle latency between adjacent tiles and packet length is up to 128 32-bit words. Bisection bandwidth equals 2660 Gbps. Due to the platform architecture, the size of FIFO buffers is limited to the power of 2. To tolerate faults, the proposed approach requires buffers of size 3. Hence, we provide communication buffers of size 4 for storing 3 packets in total: one current duplicate packet, one new main packet and one new duplicate. The platform runs at the frequency of 862.5 MHz so that one execution cycle approximately takes 1.1594 ns. The platform employs deterministic XY routing with the dead-lock and live-lock free algorithm suitable for real-time systems [13].

We have first evaluated performance of non-FT and FT parallel implementations derived from the case study Simulink model without tasks reallocation. The evaluation results have shown that the proposed FT mechanism reduces performance of the parallel code by only about 1% due to the fact that the on-chip network provides high communication bandwidth.

Moreover, we have evaluated tasks performance in the circumstances of dynamic reconfiguration. The results have

```
while (q_1 || … || q_n)
  …
  q_j = Read_from_j-th_buffer
  if ((pkt_q_j.id<=lsn) || (pkt_q_j.id<pkt_q_1.id))
    q_j = true end if
  …
end while
// Store local copy of id after reading
// packets from all input buffers
lsn = pkt_q_1.id;
```

$j \in \{1, \dots, n\}$
n – number of input buffers
$q_j = \begin{cases} \text{false, a packet has been read} \\ \text{true, read is needed} \end{cases}$
lsn – local copy of packets id
$pkt\_q_j.id$ – packet id from jth buffer
$pkt\_q_1.id$ – packet id from 1st buffer

Fig. 5. Algorithm for intelligent packet receiving in stateless regular tasks

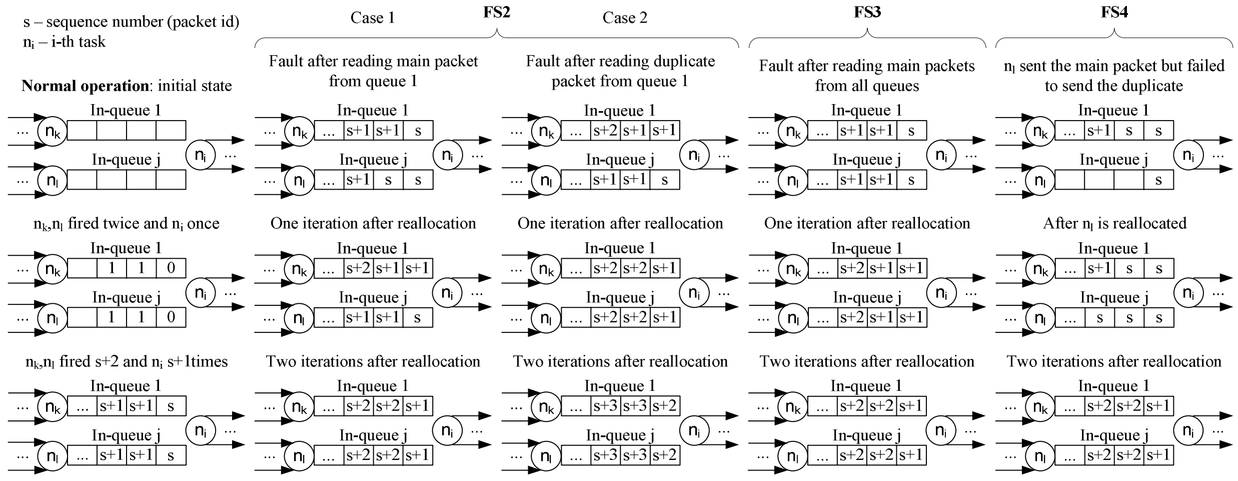

s – sequence number (packet id)
n_i – i-th task
Normal operation: initial state
In-queue 1
n_k
In-queue j
n_l
n_i
...
n_k,n_l fired twice and n_i once
1 1 0
1 1 0
n_k,n_l fired s+2 and n_i s+1times
... s+1 s+1 s
... s+1 s+1 s
FS2
Case 1
Fault after reading main packet from queue 1
... s+1 s+1 s
... s+1 s s
One iteration after reallocation
... s+2 s+1 s+1
... s+1 s+1 s
Two iterations after reallocation
... s+2 s+2 s+1
... s+2 s+2 s+1
Case 2
Fault after reading duplicate packet from queue 1
... s+2 s+1 s+1
... s+1 s+1 s
One iteration after reallocation
... s+2 s+2 s+1
... s+2 s+2 s+1
Two iterations after reallocation
... s+3 s+3 s+2
... s+3 s+3 s+2
FS3
Fault after reading main packets from all queues
... s+1 s+1 s
... s+1 s+1 s
One iteration after reallocation
... s+2 s+1 s+1
... s+2 s+1 s+1
Two iterations after reallocation
... s+2 s+2 s+1
... s+2 s+2 s+1
FS4
n_l sent the main packet but failed to send the duplicate
... s+1 s s
... s
After n_l is reallocated
... s+1 s s
... s s s
Two iterations after reallocation
... s+2 s+2 s+1
... s+2 s+2 s+1

Fig. 6. Intelligent reading in regular tasks: buffer states

illustrated that the deviation of task performance is at most 0.6% when comparing original and spare locations. In some cases, performance of the task reallocated to a spare PU has been better than the performance of the same task at the original location. This can be explained by the fact that there is lighter traffic to spare cores when routing packets.

We have also analyzed performance of the reallocation procedure. Please refer to the technical report on details [18].

VII. Conclusion and Future Work

We have shown an approach to deriving parallel programs from arbitrary discrete single-rate Simulink models. Relying on the behaviour of the resulting ACG, we have introduced a scalable FT mechanism that prevents data loss when application tasks are relocated due to failures of PUs. We have evaluated performance of the derived programs as well as of the proposed FT mechanism. The results show only about 1% performance decrease when comparing non-FT and FT versions. Thus, the proposed approach maintains efficiency and provides resilience to faults allowing applications to produce the expected result. The proposed FT can also be used separately from Simulink but requires the aforementioned assumptions. Moreover, it is not restricted to data parallel applications and can be applied to functionally parallel ones.

The future directions of our work include the development of a tool support for the proposed approach and its extension to multi-rate models. Moreover, one can integrate the proposed approach into FT dataflow proposed in [12].

Acknowledgment

The authors would like to thank Adj. Prof. Juha Plosila for fruitful discussions. The work is supported by the Digihybrid project in the EFFIMA program coordinated by FIMECC.

References

[1] Simulink, Simulation and Model-Based Design, 2014. Available: http://www.mathworks.se/help/simulink/index.html

[2] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, Computer, IEEE, Vol. 35, Issue 1, pp. 70-78, 2002.

[3] Tilera, Tile Processor User Architecture Manual, 2011. Available: http://www.tilera.com/scm/docs/UG101-user-architecture-reference.pdf

[4] F. Khalili, H. R. Zarandi, A Fault-Tolerant Low-Energy Multi-Application Mapping onto NoC-based Multiprocessors, Computational Science and Engineering, Nicosia, IEEE, pp. 421-428, 2012.

[5] G. Link, N. Vijaykrishnan, Hotspot Prevention Through Runtime Reconfiguration in Networks-on-Chip, DATE, IEEE, pp. 648-649, 2005.

[6] C. Bolchini, M. Carminati, A. Miele, Self-Adaptive Falt-Tolerance in Multi-/Many-Core Systems, Journal of Electronic Testing: Theory and Applications, Vol. 29, Issue 2, Springer US, pp. 159-175, 2013.

[7] C.-L. Chou, R. Marculescu, FARM: Fault-Aware Resource Management in NoC-based Multiprocessor Platforms, DATE Conference & Exhibition, Grenoble, IEEE, pp. 1-6, 2011.

[8] S. Ostroumov, L. Tsiopoulos, J. Plosila, K. Sere, Formal Approach to Agent-Based Dynamic Reconfiguration in Networks-On-Chip, Journal of Systems Architecture, 59(9), Elsevier, pp. 709-728, 2013.

[9] L. Guang, J. Plosila, J. Isoaho, H. Tenhunen, Hierarchical Agent Monitored Parallel On-Chip System: A Novel Design Paradigm and its Formal Specification, IJERTCS, Vol. 1, Issue 2, IGI, pp. 86-105, 2010.

[10] M. Noraziz Sham Mohd Sayuti, L. Soares Indrusiak, Real-Time Low-Power Task Mapping in Networks-on-Chip, Computer Society Annual Symposium on VLSI, IEEE, pp. 14-19, 2013.

[11] C. Pinello, L. Carloni, A. Sangiovanni-Vincentelli, Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications, International Conference on Design Automation and Test in Europe, IEEE, pp. 1164-1169, 2004.

[12] J. Ceponis, E. Kazanavicius, A. Mikuckas, Fault Tolerant Process Networks, Information Technology and Control, Vol. 35, No. 2, pp. 124-130, 2006.

[13] V. Rantala, T. Lehtonen, J. Plosila, Network on Chip Routing Algorithms, TUCS Technical Report 779, pp. 10-16, 2006.

[14] P. Boström, Contract-based verification of Simulink models, ICFEM, Durham, Springer-Verlag Berlin Heidelberg, pp. 291-306, 2011.

[15] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs and S. Tao, Implementing fail-silent nodes for distributed systems, IEEE Transactions on Computers, Vol. 45(11), pp. 1226–1238, 1996.

[16] R. Isermann, Model-based fault-detection and diagnosis – status and applications, Annual Reviews in Control 29(1), Elsevier, Vol. 29, Issue 1, pp. 71-85, 2005.

[17] L. Pike, S. Niller, N. Wegmann, Runtime Verification for Ultra-Critical Systems, In Proceedings of International Conference on Runtime Verification, Springer, pp. 310-324, 2012.

[18] S. Ostroumov, P. Boström, M. Waldén, M. Huova, Deriving Efficient and Dependable Parallel Programs from Simulink models, TUCS technical report 1111, 2014.

Paper 4

VHDL Code Generation from Formal Event-B Models

Sergey Ostroumov, Leonidas Tsiopoulos

Originally published in:
Proceedings of International Conference on Digital Systems Design (DSD), IEEE Computer Society Conference Publishing Services (CPS), pp. 127-134, 2011.

© 2011 IEEE. Reprinted, with permission, from IEEE.

2011 14th Euromicro Conference on Digital System Design

VHDL Code Generation from Formal Event-B Models

Sergey Ostroumov, Leonidas Tsiopoulos
Department of Information Technologies
Åbo Akademi University
Turku, Finland
e-mail: {Sergey.Ostroumov, Leonidas.Tsiopoulos}@abo.fi

Abstract—**In this paper, we present an approach that allows to generate VHDL code from formal models developed with the Event-B formalism. The approach is based on the relationship between the structure of the formal model and hardware description language statements. We are aiming at getting VHDL code whose behaviour is the same as the behaviour of the Event-B model. Our contribution lies in the fact that we show the main similarity between the formal model and VHDL code that allows us to derive the method and, hence, the algorithm for automatic translation. This algorithm can be implemented as a plug-in for the Rodin tool which supports the Event-B formalism. The approach is presented through a simplified version of an industrial case study developed in a stepwise refinement manner. We also present several ways of possible translation depending on the way the model has been developed through refinement. In addition, we present synthesis results that show space occupied by the VHDL code generated.**

Keywords-formal modelling; Event-B; VHDL; code generation

I. Introduction

Advances in technology allow us to design embedded systems as single chip systems, so-called Systems-On-Chip. On one hand, it allows us to reduce the space and power consumption and diminish time delays in a system. On the other hand, the complexity of such type of systems leads to impossibility of performing exhaustive testing in order to ensure the correctness of the system.

The first solution to this problem may be found by applying the model-checking approach to implementations [1]. The main idea of this approach is to derive a model of a system from the system implementation (code) and then check system properties on that model. However, this approach is time consuming because if errors are found, then it is necessary to return to the implementation, correct the errors and apply model-checking repeatedly until the correct system is derived.

Another way to solve the problem is offered with a formal model development of a system [2]. A model can be developed in a stepwise manner which is known as the *refinement* approach. At every step of the refinement-based approach we introduce functional properties of the system that have to always be established while the model is being developed. These properties are called *invariants*. Each refinement step has to preserve the invariants mentioned at a particular step and all steps before. Therefore, the formal model of the system is correct by construction. In other words, we can guarantee that the behaviour of the system matches the requirements.

There exist several formal methods for hardware specification and verification such as Action Systems [3], Lustre [4], Signal [5] and Esterel [6]. The synchronous formalisms Lustre, Signal and Esterel cannot model both synchronous and asynchronous systems in a straightforward manner because of the *perfect synchrony hypothesis*, in which outputs are produced synchronously with the inputs. Specifically, Signal has been applied to modelling globally asynchronous designs in synchronous networks [7] while Esterel has been extended to model multiple clock domains [8]. Moreover, Halbwachs and Baghdadi [9] introduced several extensions in order to avoid the limitations of the perfect synchrony hypothesis. These extensions add significant overhead to the specifications. Furthermore, only Signal supports the refinement approach.

For the work in this paper we focus on the Event-B formalism [2], related to Action Systems and its extension B Action Systems [10], because it has adequate tool support enabling system-level modelling and analysis of both synchronous and asynchronous systems. It allows us to model a discrete transition system and prove the consistency of the model following the refinement approach. In addition, we can model concurrency by atomic events that can be executed in parallel if they operate on disjoint variables. This approach is supported by the Rodin platform [11], which is open source software and allows us to extend its functionality by adding new modules in the form of *plug-ins*.

The target language we are aiming at is VHSIC Hardware Description Language (VHDL) [12]. This language is standardized [13] and is widely used in hardware design to describe systems based on field-programmable-gate-arrays. There are a number of tools that support VHDL designs and allow us to synthesize the code and simulate its behaviour. In order to do that, we have used Quartus-II software for synthesizing the code and ModelSim software for simulating this code [14].

Since a formal model is correct by construction, it is very important to start with the formal modelling of the system and then transform the model into a programming language. We are aiming at the creation of a plug-in that makes the translation process automatic and the development of the method for generation of the VHDL code from the given Event-B model that the plug in implements. The method is based on the correspondence between a formal model and VHDL that enables the automatic translation to be performed correctly.

978-0-7695-4494-6/11 $26.00 © 2011 IEEE
DOI 10.1109/DSD.2011.20

IEEE computer society

The rest of the paper is organised as follows. Section II describes related work and compares our approach to other approaches. In section III, we describe the structure of an Event-B model and present formal proof obligations that a model has to be consistent with. The structure and statements of VHDL code are depicted in section IV. The approach we use to derive VHDL code from a model is represented in section V. In section VI we show an application of this approach with a simplified version of an industrial case study. Section VII discusses possible extensions of the approach if a model is decomposed through refinement. We discuss future work and directions of the research in section VIII.

II. Related work

The work presented by Cansell, Méry and Proch [15] introduces the approach to transformation of the formal model into SystemC language. The authors aim at the formalization of the SystemC scheduler which handles timing requirements of a program in this language. However, this approach cannot be applied to transformation into more low level language, namely hardware description language (HDL), because HDL does not have any schedulers as such.

An approach presented by Plosila and Sere [16] has been developed for modelling and verification of asynchronous hardware systems. In this work the authors describe the formal design process for an asynchronous pipelined processor that contains concurrent elements. This approach relies on the use of the Action Systems formalism. The application of the Action Systems formalism to synchronous systems has been extended by Seceleanu [17]. This approach relies on the two-phase operation (read and write) of a synchronous system modelled with the Action Systems formalism while our approach focuses on facilitating the translation of implementable hardware models developed within Event-B into a targeted hardware description language.

An approach for transformation of formal models to VHDL has been developed by Hallerstede and Zimmermann [18]. This approach lies in the field of transforming a linear system into HDL. In fact, this approach is used by the AtelierB tool [19] which is based on the B Method formalism and supported by industrial partners [20]. This approach uses a middleware language called B0 and then allows us to get pure VHDL. Since Event-B is an extension of B Method [21] which allows us to model reactive systems, it is not straightforward how to apply this approach to the Event-B formalism. Besides, our approach considers Event-B models as such and allows us to generate VHDL code directly from a formal model.

III. Event-B

Event-B is a formal method based on correct-by-construction development of systems through stepwise refinement. To start with, we have to describe some important structures of an Event-B model [2]. The model consists of two parts: a context and a machine. The context contains static elements of the model such as sets, constants and axioms that can be seen by the machine. The machine consists of model properties (invariants) and dynamic constituents (variables and events).

We do not give the structure of the context because from the relation between machines and contexts we can get all the required values. We focus on the detailed description of the dynamic part of a model (machine) to show the correspondence between a formal model and VHDL code.

The overall structure of the machine is given in Fig. 1.

```
machine Machine
sees Context
variables
//Global variables of a model
invariants
//Variables type and properties of a model
events
//Actions that model performs
end
```

Figure 1. The structure of an Event-B machine.

There are five main sections in a machine. As any component, the machine has a label (its name), which we represent by using the **machine** keyword. If we have defined static properties in a context, then they are seen through the **sees** clause. Since the machine is the dynamic part, it changes its state by modifying variables introduced in the **variables** clause. In the **invariants** clause we give the types of the variables and the guaranteed state properties of the machine. The behaviour of the machine can be modelled by **events** specified in the corresponding clause.

The events have the following structure which we are using for proper translation of the model to the code. The syntax of an event is as follows:

$$\mathrm{Evt} = \mathrm{WHEN}\ \mathrm{g}\ \mathrm{THEN}\ \mathrm{S}\ \mathrm{END}$$

where Evt is an event, g represents the conjunction of guards (conditions on which the event is enabled to be executed) and S is a statement defined as an assignment to the variables.

In the model we can use two types of assignments: deterministic and non-deterministic. Since a hardware system must have a deterministic behaviour, the final model must have only deterministic assignments of the following form: $\mathrm{x} := \mathrm{Exp}$, where x reflects the vector of the variables and Exp is an expression.

Formally, each event is viewed as a before-after predicate $\mathrm{BA}(x,x')$ that links the values of the variables just before (x) and just after (x′) of the event execution. This scheme makes it possible to prove the correctness of the model by preserving the invariants. A model is consistent if the following proof obligations hold:

1. $\mathrm{WD}(\mathrm{Inv})$
2. $\mathrm{BA}(x{:=}x0) \Rightarrow \mathrm{Inv}'$
3. $\mathrm{G}_i \wedge \mathrm{Inv} \wedge \mathrm{BA}_i(\mathrm{S}) \Rightarrow \mathrm{Inv}'$

where 1. shows well-definedness [22] of an invariant, 2. depicts the establishment of this invariant at initialisation and 3. states that every event preserves the invariant. Inv' stands for a modified invariant containing the updated state variables after an event execution.

Since a model is developed in a stepwise manner, the following proof obligations have to be consistent with respect to the refinement approach:

1. $\mathrm{AbsInv} \wedge \mathrm{ConcInv} \wedge \mathrm{ConcG} \Rightarrow \mathrm{AbsG}$
2. $\mathrm{AbsInv} \wedge \mathrm{ConcInv} \wedge \mathrm{ConcG} \wedge \mathrm{ConcBA} \Rightarrow \mathrm{AbsBA}$
3. $\mathrm{ConcG} \wedge \mathrm{ConcBA} \Rightarrow \mathrm{AbsBA}$

where 1. corresponds to guard strengthening, 2. shows simulation of an action and 3. represents the equality of the preserved ("old") variable. AbsInv, AbsG and AbsBA stand for an invariant, guard and before-after predicate of an event that appear in a previous refinement step while ConcInv, ConcG and ConcBA reflect an invariant, guard and before-after predicate of an event that are stated at the current refinement step.

The whole Event-B model is developed in a stepwise manner following a refinement-based approach. At every refinement step, we add details of a system. In addition, we prove the consistency between a more abstract specification and a more concrete model by preserving the invariants from the previous steps.

Thus, every model which is developed in this manner has logical proofs of its consistency with the respect to the properties derived from the system requirements. In other words, Event-B offers a proof-based verification. Nevertheless, a model checker and animator (ProB [23]) is also available.

IV. VHDL

Similarly to the Event-B models, we have to present some VHDL constructions that are useful for the translation process. We begin with describing the structure that a VHDL project has [12, 13].

The starting point in VHDL code is a clause named **entity** (Fig. 2). Every entity must have a name and some ports. Additionally, we can define necessary parameters for the entity by using **generic** statement. Every input and output of the entity is introduced in the **port** clause. All the signals in the entity represent the interface of this entity and have a direction and a type. Thus, we introduce the **in** direction for the input signals and the **out** direction for the output signals. The usual type for every signal is **std_logic** independent of signal direction.

```
entity Entity is
generic (
--Parameters);
port
(--Inputs : in std_logic,
--Outputs : out std_logic);
end Entity;

architecture arch of Entity is
--Internal signals
begin
//Statements
end
```

Figure 2. The structure of an entity in VHDL.

Using the interface described in the entity, we implement the hardware behaviour in the **architecture** clause. *Internal signals* of the architecture depict the internal data of the entity. Since an action modifies the value of the variables, the assignment to signals in terms of VHDL has the following form: x <= E, where x is a signal and E is an expression. Every such an assignment is not instant. In other words, every signal has a buffer which contains the value after the assignment. Hence, it is not possible to perform several assignments to one signal unless this signal is in the process clause. Thus, we focus on the **process** clause where we use the "**if** condition **then** action **end if**" statement whose behaviour is the same as the behaviour of an event in the model. The syntax defining the process is as follows:

process (<*sensitivity list*>) **is begin** *operators* **end**;

where <*sensitivity list*> is the list of signals on which the process is activated and for *operators* we use the conditional statement described above.

Therefore, we have derived the main similarity between the Event-B model and VHDL code.

V. Event-B Definitions In Terms Of VHDL Statements

The correspondence between a machine and an entity is depicted in Fig. 3 and Fig. 4.

Since a machine may have different variables that represent the inputs and outputs as well as internal data, there is a necessity to distinguish them. In order to do that, all the variables that have "_I" suffix are considered to be the inputs, the variables with "_O" suffix represent the outputs, while the others reflect the internal data. All the variables have to be of the numeric or Boolean type. The correspondence of the types is given in Tab. I

An Event-B model can be developed in different ways with respect to the refinement approach. Usually, we start with a very abstract model that has non-deterministic behaviour and we come to completely deterministic actions while refining the model. The final deterministic model can be translated into asynchronous VHDL code.

Event-B structures	VHDL statements
machine *Machine* **variables** *//Inputs Outputs Internal* **invariants** *//Variables type* **events** **event** INITIALISATION **begin** *//Default values* *//on the variables* **end**	**entity** *Machine* **is** **port** *(--Inputs and outputs, their type* *--and the default value, e.g.* *Inputs :* **in std_logic :=** *def_val,* *Outputs :* **out std_logic :=** *def_val* *);* **end** *Machine;*

Figure 3. Correspondence between a machine and an entity.

In order to derive the VHDL code, we use the following correlation between the structure of Event-B events and VHDL statements (Fig. 4).

TABLE I. Types Correspondence

Event-B types	VHDL types
BOOL *= {TRUE,FALSE}*	Std_logic *= {1,0}*
n..m, *where n, m – numbers*	Integer range n to m

As it can be seen, every event of the model is reflected by the corresponding "**if then end if**" statement with the same name. The guards of the event are translated to be the conditions of the statement and the event actions are the assignments in terms of VHDL. There is one event which needs to be translated in a different way. This event models the non-deterministic assignment on the inputs which distinguishes it from other events, i.e. this event models the environment. The variables that appear in a guard of this event and have a deterministic assignment on them are translated in accordance to the structure described above. If there are some variables that depend on new values of the inputs, then this assignment is translated as an additional "if" statement.

Event-B structures	**VHDL statements**
events	**architecture** a **of** Entity **is begin**
event *evt1*	*//Internal variables*
when	**process** (*variables*) **is begin**
@grd1 *G*	*evt1*:
then	**if** *G*
@act1 *A*	**then** *A*
end	**end if;**
end	**end process;**
	end a

Figure 4. Correlation between events and architecture.

To enable a smooth translation of the model into VHDL, the model has to be deterministic. In other words, the last refinement step has to contain only deterministic assignments on the variables that have presented types.

Therefore, having the final Event-B model (the final refinement) we transform this model to VHDL code by following these rules.

The translation algorithm contains the following steps:

1. Every variable is transformed into signals that have the type defined in Tab. 1.
2. All variables with suffixes "_I" and "_O" are put into port clause of the entity and are provided with corresponding keyword that reflects the direction of each signal (in or out).
3. The other variables are represented as internal signals and are put into architecture clause.
4. Since all the variables are initialized, so are the signals, independently of the clause they are put in.
5. The process clause is added to the architecture with the sensitivity list containing all the signals.
6. Every event of the model is transformed into "if then" statement that hat the corresponding label.
7. The event that models an assignment to inputs is translated as a special "if" statement that changes the value of the signals appeared in the condition.

The case study in the next section presents the application results of this approach.

VI. Case Study

The case study is a simplified version of one of the avionics systems which was developed in collaboration with the SSPE "Corporation Kommunar" ST SCB "Polisvit" [24]. This industrial partner specialises on the development of control systems for avionics and space. One of those systems was taken as a basis for the case study.

The main goal of the system is to prevent the moving parts of a plane from being covered with ice. In order to perform this function, the system turns heaters on under the following conditions: either a pilot switches on the system or the sensor detects ice on the moving parts of an airplane.

There are two types of heaters in the system: one that is switched on constantly and another one that has a cyclic behaviour. The system consists of one heater of the first type and three heaters of the second type. Every heater of the second type is turned on one-by-one because it is forbidden for the controller to turn on several heaters of the second type simultaneously.

We can now concentrate on the formal model of the system and its translation into VHDL code. We first present the initial formal specification and then we present the new features introduced in each refinement. The final refinement is the starting point for code generation.

A. Initial Specification of the Case Study

We start modelling the system as a "black box". In other words, we introduce all the inputs and outputs of the system, but we do not specify all the algorithmic details. The initial specification of the case study is given in Fig. 5. All variables introduced in this step are of the BOOL type and the initial value that is assigned to all variables is FALSE.

The safety invariants for this specification describe the dependencies between inputs and outputs. For instance, it is impossible for the system to have two input signals being TRUE at the same time. In this system, there are two main inputs that influence the mode of the system. They are reflected by the variables Manual_I and Auto_I. Manual_I equals to TRUE when a pilot switches the system on manually and Auto_I enables the automatic mode when the system detects ice. Hence, it is impossible to turn these two modes on simultaneously. These conditions are depicted by inv0_9 together with the thm2 and thm3. Furthermore, inv0_10 and inv0_11 state that the heaters are on under the above conditions on the inputs. Otherwise, they are off.

Abstractly, the system reads the inputs, and produces the outputs (which are "On"/TRUE or "Off"/FALSE in this system) depending on the inputs read. These activities are modelled by the corresponding events, namely the **Read_inputs** event assigns some values to the inputs, the **Heaters_OFF** event reflects the turning off of the heaters and the **Heaters_ON** event modifies the outputs such that the heaters are on.

B. The First Refinement of the Case Sudy

In the first refinement step, we focus on the heaters – the outputs of the system. The system has heaters that cannot be switched on simultaneously. To specify this requirement, we introduce a new variable, named Iteration, as well as safety invariants stating impossibility of switching the heaters on concurrently. The variable is initialised to 0 and incremented each time when one of the heaters is turned on. The new variable and invariants are depicted in Fig. 6.

Since these heaters are on one-by-one, the invariants inv1_1 and inv1_2 connect the iterations with switching on the heaters. In addition, while one of the heaters is on, the others must be off. This dependency is reflected by inv1_4, inv1_5 and inv1_6. These invariants guarantee the deterministic behaviour of the heaters.

```
machine M0_IPS
variables
          AbsMode, Auto_I, Manual_I, Ice_I, Heater1_O, Heater2_O,
          Heater3_O, Heater_Knife_O
invariants
             // All the variables have BOOL type in the abstract specification
          inv0_9   :          Auto_I=FALSE ∨ Manual_I=FALSE
                              AbsMode = FALSE ⇒ ((Manual_I = TRUE ∨
          inv0_10 :           (Auto_I = TRUE ∧ Ice_I=TRUE)) ⇔ ((Heater1_O=
                              TRUE ∨ Heater2_O=TRUE ∨ Heater3_O=TRUE)))
                              AbsMode = FALSE ⇒ (¬(Manual_I = TRUE ∨
          inv0_11  :          (Auto_I = TRUE ∧ Ice_I=TRUE)) ⇔ ((Heater1_O=
                              FALSE ∧ Heater2_O=FALSE ∧ Heater3_O=FALSE)))
                              (Manual_I = TRUE ∨ (Auto_I = TRUE ∧
          thm1   :            Ice_I = TRUE)) ∨ (Manual_I = FALSE ∧
                              Auto_I = FALSE) ∨ (Auto_I = TRUE ∧ Ice_I = FALSE)
          thm2   :            Manual_I = TRUE ⇒ Auto_I = FALSE
          thm3   :            Auto_I = TRUE ⇒ Manual_I = FALSE
events
INITIALISATION  ≙
BEGIN // All the variables are initialised to FALSE END
Read_inputs  ≙                      // The system reads the inputs
WHEN
            grd1  :        AbsMode = FALSE
THEN
            act1  :        AbsMode ≔ TRUE
                           Auto_I,Manual_I, Ice_I :| Auto_I' ∈ BOOL ∧
            act2  :        Manual_I' ∈ BOOL ∧ Ice_I'∈BOOL ∧
                           (Auto_I'=FALSE ∨ Manual_I'=FALSE)
END
Heaters_OFF  ≙
WHEN
            grd1  :        AbsMode = TRUE
            grd2  :        ¬(Manual_I = TRUE ∨ (Auto_I=TRUE ∧ Ice_I = TRUE))
THEN
            act1  :        AbsMode ≔ FALSE
                           Heater1_O,Heater2_O,Heater3_O,
            act2  :        Heater_Knife_O ≔ FALSE,FALSE,FALSE,FALSE
END
Heaters_ON  ≙
WHEN
            grd1  :        AbsMode = TRUE
            grd2  :        (Manual_I = TRUE ∨ (Auto_I=TRUE ∧ Ice_I = TRUE))
THEN
            act1  :        AbsMode :∈ BOOL
                           Heater1_O,Heater2_O,Heater3_O :|
            act2  :        Heater1_O'∈BOOL ∧ Heater2_O'∈BOOL ∧
                           Heater3_O'∈BOOL ∧ (Heater1_O'=TRUE ∨
                           Heater2_O'=TRUE ∨ Heater3_O'=TRUE)
            act3  :        Heater_Knife_O ≔ TRUE
END
END
```

Figure 5. The initial specification of the case study.

```
variables
      ..., Iteration
invariants
inv1_1  :  Iteration ∈ 0⍰Max_iter
           (AbsMode = TRUE ∧ Iteration=2) ⇒
inv1_2  :  (Heater1_O = TRUE ∧ Heater2_O = FALSE ∧ Heater3_O = FALSE)
           (AbsMode = TRUE ∧ Iteration=3) ⇒
inv1_3  :  (Heater1_O = FALSE ∧ Heater2_O = TRUE ∧ Heater3_O = FALSE)
inv1_4  :  Heater1_O = TRUE ⇒ Heater2_O = FALSE ∧ Heater3_O = FALSE
inv1_5  :  Heater2_O = TRUE ⇒ Heater1_O = FALSE ∧ Heater3_O = FALSE
inv1_6  :  Heater3_O = TRUE ⇒ Heater1_O = FALSE ∧ Heater2_O = FALSE
```

Figure 6. The first refinement. New variables and invariants.

C. The Second Refinement of the Case Sudy

In the second refinement step, we introduce new variables and invariants that reflect different properties of the system (Fig. 7).

The TimeCnt variable models the counter that counts how long a heater is on. This counter is activated when the TimeCnt_Ena variable has the value TRUE. While the counter is enabled (i.e. TimeCnt_Ena = TRUE), one of the cyclic heaters is turned on (inv2_3). In addition, when the counter stops, it resets its value (inv2_4).

```
variables
      ..., TimeCnt, TimeCnt_Ena, TimeReached, Cmp
invariants
inv2_1  :     TimeCnt ∈ 0⍰WordSize
inv2_2  :     TimeCnt_Ena ∈ BOOL
              TimeCnt_Ena=TRUE ⇒
              ((Heater1_O=TRUE ∧ Heater2_O=FALSE ∧
inv2_3  :     Heater3_O=FALSE)∨(Heater1_O=FALSE ∧
              Heater2_O=TRUE ∧ Heater3_O=FALSE)∨
              (Heater1_O=FALSE ∧ Heater2_O=FALSE ∧ Heater3_O=TRUE))
inv2_4  :     TimeCnt_Ena = FALSE ⇒ TimeCnt = 0
```

Figure 7. The next refinements: new variables and invariants.

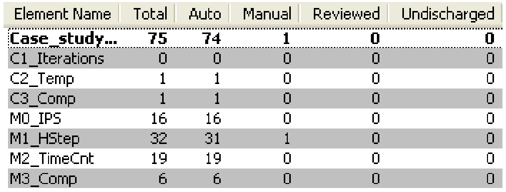
D. The Last Refinement of the Case Sudy

The counter cannot count infinitely long. Therefore, there exists an upper bound for counting. This bound is introduced in the last refinement step (Fig. 8). When this bound is reached the value of TimeReached variable becomes TRUE. In order to modify this value, every increment on TimeCnt is compared to the upper bound. If the Cmp variable has TRUE value, the comparator compares the current value of the counter with the upper limit. If these values are equal, then the counter has reached the limit (inv3_3).

```
machine M3_Comp
refines M2_TimeCnt
sees C3_Comp
variables
   AbsMode, Auto_I, Manual_I, Ice_I, Heater1_O, Heater2_O,
Heater3_O,
   Heater_Knife_O, Iteration, TimeCnt, TimeCnt_Ena, TimeReached,
Cmp
invariants
...
inv3_1  :          TimeReached ∈ BOOL
inv3_2  :          Cmp ∈ BOOL
                   (TimeCnt = TimeLimit ∧ Cmp = FALSE) ⇒
inv3_3  :          TimeReached = TRUE
events
INITIALISATION  ≙
extended
BEGIN
        // All variables of the BOOL type are initialised with FALSE
      act1_1  :        Iteration ≔ 0
      act2_1  :        TimeCnt ≔ 0
END
```

Figure 8. The model variables, their types and initialization.

Before we present the translation of the formal model to corresponding VHDL code, we depict the statistics of proof obligations for the case study in Fig. 9.



Element Name	Total	Auto	Manual	Reviewed	Undischarged
Case_study...	**75**	**74**	**1**	**0**	**0**
C1_Iterations	0	0	0	0	0
C2_Temp	1	1	0	0	0
C3_Comp	1	1	0	0	0
M0_IPS	16	16	0	0	0
M1_HStep	32	31	1	0	0
M2_TimeCnt	19	19	0	0	0
M3_Comp	6	6	0	0	0

Figure 9. Number of proof obligations.

E. Translation of the Last Refinement to VHDL Code

The last refinement of the system contains all the variables with their types and initial values which can be translated into VHDL (Fig. 8). As it can be seen, all the variables have either BOOL type or Integer type with the range from 0 to some constant. These constants reflect the parameters of the model. The constant Max_iter represents the maximum number of iterations and the WordSize defines the maximum value of a counter. These constants are introduced in the contexts of the model and have type ℕ. The Max_iter equals to 4 and the WordSize takes a value of 2 to the power 2 (2^2).

All the constants of the model appear as the parameters of the entity in the generic clause (Fig. 10). All variables with the "_I" suffix depict the inputs of the system while the variables with "_O" suffix are the outputs. The others correspond to internal signals in the architecture.

Since all the variables in the model are initialised, so are the signals in the VHDL code. Besides, the input and internal signals are in the "sensitivity list" of the process that defines the behaviour of the entity.

To be able to use types and operations on them, the translation process starts with adding library "ieee" with all necessary branches. This library is commonly used in hardware design.

Compared to the initial specification, the last refinement contains eight events, namely **Read_inputs**, **Heaters_OFF**, **Heaters_ON1**, **Heaters_ON2**, **Heaters_ON3**, **TimeCount**, **Comparator**, **TimeStop**. We show several examples of the events translation to the corresponding "if" statements.

The event depicted in Fig. 11 models the switching the heaters off.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY M3_Comp IS
GENERIC
(           Max_iter    :           INTEGER := 4;
            WordSize    :           INTEGER := 2**4;
            TimeLimit   :           INTEGER := 2**2);
PORT
(           --Input ports
            Auto_I       : IN        STD_LOGIC := '0';
            Manual_I     : IN        STD_LOGIC := '0';
            Ice_I        : IN        STD_LOGIC := '0';
            --Output ports
            Heater1_O    : OUT       STD_LOGIC := '0';
            Heater2_O    : OUT       STD_LOGIC := '0';
            Heater3_O    : OUT       STD_LOGIC := '0';
            Heater_Knife_O          : OUT        STD_LOGIC := '0');
END M3_Comp;
ARCHITECTURE a OF M3_Comp IS
SIGNAL AbsMode        :  STD_LOGIC := '0';
SIGNAL Iteration      : INTEGER RANGE 0 TO Max_iter := 0;
SIGNAL TimeCnt        : INTEGER RANGE 0 TO WordSize := 0;
SIGNAL TimeCnt_Ena    : STD_LOGIC := '0';
SIGNAL TimeReached    : STD_LOGIC := '0';
SIGNAL Cmp            : STD_LOGIC := '0';
BEGIN
            M3_Comp:
            PROCESS (Auto_I,Manual_I,Ice_I,AbsMode,Iteration,
            TimeCnt,TimeCnt_Ena,TimeReached,Cmp) IS BEGIN
```

Figure 10. VHDL interpretation of the signals, their types and default values.

```
Heaters_OFF  ≜
extended
REFINES
      Heaters_OFF
WHEN
      grd1  :  AbsMode = TRUE
      grd2  :  ¬(Manual_I = TRUE ∨
               (Auto_I=TRUE ∧ Ice_I = TRUE))
THEN
      act1  :  AbsMode ≔ FALSE
      act2  :  Heater1_O,Heater2_O,Heater3_O,Heater_Knife_O ≔
               FALSE,FALSE,FALSE,FALSE
      act   :  TimeCnt ≔ 0
      act4  :  TimeCnt_Ena ≔ FALSE
      act5  :  TimeReached ≔ FALSE
END
```

Figure 11. Switch off the heaters event.

Since the multiple assignments are not supported by VHDL, the second action of the event is split up into four separate assignments (Fig. 12).

```
Heaters_OFF:
IF (AbsMode = '1') and
  (not(Manual_I = '1' or (Auto_I='1' and Ice_I = '1')))
THEN        AbsMode <= '0';
            TimeCnt <= 0;
            TimeCnt_Ena <= '0';
            TimeReached <= '0';
            Heater1_O <=  '0';
            Heater2_O <= '0';
            Heater3_O <= '0';
            Heater_Knife_O  <= '0';
END IF;
```

Figure 12. VHDL interpretation of the switch off the heaters event.

The **Read_inputs** event represents the specific event that belongs to the environment. This event has variables that behave deterministically, namely AbsMode, TimeCnt, TimeCnt_Ena and TimeReached. The variable Iteration also has a deterministic behaviour, although it is modified depending on the new values on the inputs (Fig. 13).

```
Read_inputs  ≜          //  The system reads the inputs
extended
REFINES
      Read_inputs
WHEN
      grd1  :   Ab Mode = FALSE
      grd2  :   TimeCnt = 0
      grd3  :   TimeCnt_Ena = FALSE
THEN
      act1  :   AbsMode ≔ TRUE
                Auto_I,Manual_I, Ice_I, Iteration :|
                Auto_I' ∈ BOOL ∧ Manual_I' ∈ BOOL ∧ Ice_I' ∈ BOOL ∧
      act2  :  Iteration'∈0..Max_iter ∧ (Auto_I'=FALSE ∨ Manual_I'=FA LSE)∧
                ((Manual_I'=TRUE ∨ (Auto_I'=TRUE ∧ Ice_I'=TRUE)) ⇒
                Iteration'=1)∧(¬(Manual_I'=TRUE ∨ (Auto_I'=TRUE ∧
                Ice_I'=TRUE) ⇒ Iteration'=0))
      act3  :   TimeReached ≔ FALSE
END
```

Figure 13. The read inputs event.

These variables are translated into corresponding VHDL code (Fig. 14) in the same manner as other events. Since the values of the inputs come from the environment, there are no assignments on them in the VHDL code. The dependency of Iteration on the inputs generates an additional "if" statement.

```
Read_inputs:
IF (AbsMode = '0') and
  (TimeCnt = 0) and
  (TimeCnt_Ena = '0')
THEN        AbsMode <= '1';
            IF (Manual_I='1' or (Auto_I='1' and Ice_I='1'))  THEN
                        Iteration<=1;
            ELSIF not(Manual_I='1' or (Auto_I='1' and Ice_I='1'))  THEN
                        Iteration<=0;
            END IF;
            TimeReached <= '0';
END IF;
```

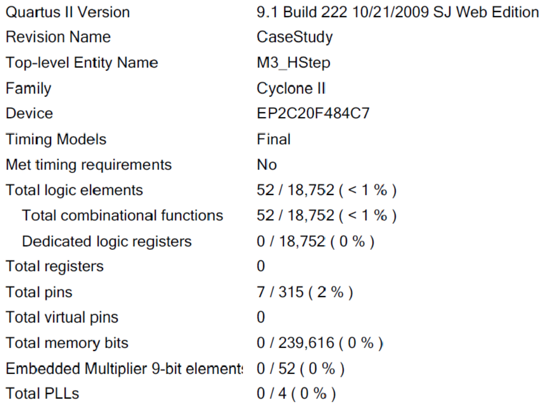
Figure 14. VHDL interpretation of the reading event.

The code achieved from the model has been synthesized and simulated.

F. *Synthesis results and behaviour comparison*

To synthesise the code, we have used the Quartus-II web-edition software which is free of charge. This software shows the statistical information about the use of the different elements of the chip chosen (Fig. 15). It also produces the firmware file that can be uploaded into a chip. We opt for Cyclone II family chips because a development board sold by Altera company is based on this chip.

In order to compare the behaviour of the formal model and the code, we used ProB tool [21] which allows us to animate ("run") the model and ModelSim tool [11] that simulates the code. A result of the animation at an execution point is shown in Fig. 16. The corresponding simulation result is depicted in Fig. 17. Similarly one can observe and compare animations and simulations for each execution step. As it can be seen from the animation and simulation results the behaviours of the model and the code are the same.

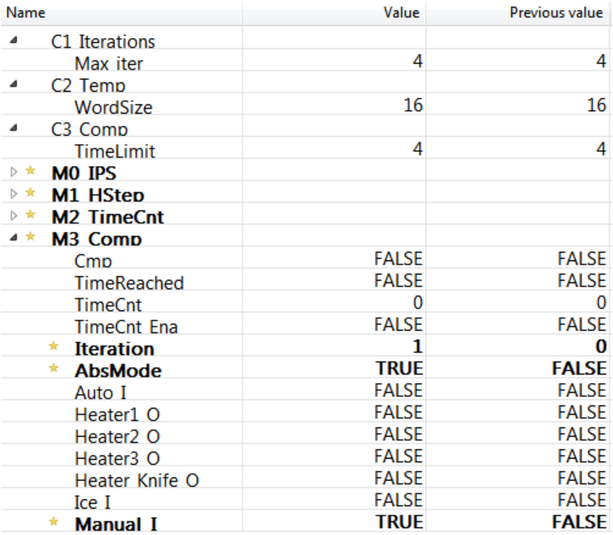


Quartus II Version	9.1 Build 222 10/21/2009 SJ Web Edition
Revision Name	CaseStudy
Top-level Entity Name	M3_HStep
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Met timing requirements	No
Total logic elements	52 / 18,752 (< 1 %)
Total combinational functions	52 / 18,752 (< 1 %)
Dedicated logic registers	0 / 18,752 (0 %)
Total registers	0
Total pins	7 / 315 (2 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit element:	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 15. Synthesis results for the generated code.

VII. Decomposition Of Event-B Models

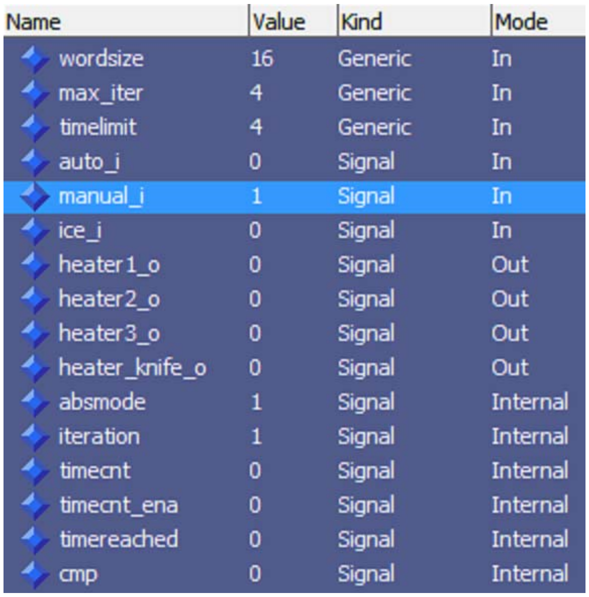
During refinements the model can become complex and difficult to read. To cope with this problem, the model can be decomposed using the shared-variable [25] or the shared-event [26] approach. The main goal of these approaches is to use a shared structure (a variable or an event) that allows us to introduce an interface between machines and, hence, the models. Our interest lies in the decomposition using shared-variables approach. In this case, the shared variables are represented as the signals between modules in VHDL and every machine is interpreted as a separate vhdl file. In accordance to this scheme, we can get a hierarchy of the models which is reflected in the VHDL code correspondingly. Certainly, while translating the models into the code, we have to introduce a top-level entity in VHDL that joins all the modules into one project.



Name	Value	Previous value
C1 Iterations		
Max iter	4	4
C2 Temp		
WordSize	16	16
C3 Comp		
TimeLimit	4	4
M0 IPS		
M1 HStep		
M2 TimeCnt		
M3 Comp		
Cmp	FALSE	FALSE
TimeReached	FALSE	FALSE
TimeCnt	0	0
TimeCnt Ena	FALSE	FALSE
Iteration	**1**	**0**
AbsMode	**TRUE**	**FALSE**
Auto I	FALSE	FALSE
Heater1 O	FALSE	FALSE
Heater2 O	FALSE	FALSE
Heater3 O	FALSE	FALSE
Heater Knife O	FALSE	FALSE
Ice I	FALSE	FALSE
Manual I	**TRUE**	**FALSE**

Figure 16. Animation result.

For instance, a model can be decomposed in such a way that there are three machines: one that models one module, another one that reflects another module and the one that connects these two with each other. Then, the translation process may proceed as depicted in Fig. 18. The "joining" machine corresponds to the top-level entity in VHDL that maps signals of the modules defined by the other entities.



Name	Value	Kind	Mode
wordsize	16	Generic	In
max_iter	4	Generic	In
timelimit	4	Generic	In
auto_i	0	Signal	In
manual_i	1	Signal	In
ice_i	0	Signal	In
heater1_o	0	Signal	Out
heater2_o	0	Signal	Out
heater3_o	0	Signal	Out
heater_knife_o	0	Signal	Out
absmode	1	Signal	Internal
iteration	1	Signal	Internal
timecnt	0	Signal	Internal
timecnt_ena	0	Signal	Internal
timereached	0	Signal	Internal
cmp	0	Signal	Internal

Figure 17. Simulation result.

Nevertheless, the translation method remains the same: a machine represents an entity with an architecture and all corresponding variables, events and statements.

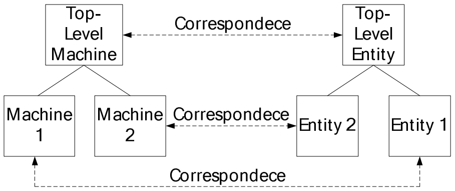


Figure 18. Decomposition of the models and its reflection in VHDL.

VIII. Conclusion And Future Work

In this paper we presented an approach that enables the translation from formal models developed in terms of the Event-B formalism into VHDL code. This approach is based on the similarities found between a formal model and the hardware design. Furthermore, the translation process that relies on this approach can be automated by using the scalability of the Rodin tool that allows us to extend its functionality with plug-ins. Currently, we have implemented a prototype version of the VHDL code generation plug-in in our department. In the near future we intend to make it available through the Rodin platform.

One direction for future research is the decomposition of Event-B models which gives us an opportunity to construct a hierarchical structure of a model so that this structure will be reflected in the hardware code. Considering this structure we will investigate the optimisation of the VHDL code, for example the use of different hardware library components in order to increase performance of the target product.

Another part of the future research is the introduction of the clock signal into a formal Event-B model. A variable that models clock present in the last refinement will not change the result of the translation process because the algorithm of the VHDL process statement does not change; the process will execute, for example, on every rising edge of the clock. Furthermore, the introduction of clock may depend on the way the model is decomposed and refined.

These future directions require the deployment of our approach on more case studies which will also enable the investigation of various metrics such as performance and used hardware resources.

In addition, we will investigate approaches to fully automate the comparison between the behaviour of a formal model and the behaviour of VHDL code.

Acknowledgment

The authors would like to thank Professor Kaisa Sere for fruitful discussions and Linas Laibinis for valuable feedback on the models.

References

[1] E. Clarke, Model Checking, Cambridge: The MIT Press, 2002.

[2] J.-R. Abrial, Modeling in Event-B. System and Software Engineering, Cambridge: Cambridge University Press, 2010.

[3] R.-J. Back, R. Kurki-Suonio, Decentralization of Process Nets with Centralized Control. In proceedings of the 2nd ACM SIGAST-SIGOPS Symposium on Principles of Distributed Computing, p. 131-142, 1983.

[4] P. Caspi, N. Halbwachs, D. Pilaud, J. Plaice, Lustre: A declarative language for programming synchronous systems, Proc. 14th POPL Symposium, 1987, p. 178-188.

[5] A. Benveniste, P. Le Guernic, Hybrid Dynamical Systems Theory and the Signal Language, IEEE Transactions on Automatic Control 35(5), 1990, p. 535-546.

[6] G. Berry, L. Cosserat, The Esterel Synchronous Programming Language and its Mathematical Semantics, In Seminar of Concurency, Lecture Notes in Computer Science vol. 197, 1985, p. 389-448.

[7] M. R. Mousavi, P. Le Guernic, J.-P. Talpin, S. Shukla, T. Basten, Modelling and Validating Globally Asynchronous Frameworks, Proc. Design Automation and Test in Europe, 2004, IEEE Computer Society Press, p. 384-389.

[8] G. Berry, E. Sentovich, Multiclock Esterel, Lecture notes in Computer Science, Correct Hardware Design and Verification Methods v. 2144, 2001, p. 110-125.

[9] N. Halbwachs, S. Baghdadi, Synchronous Modelling of Asynchronous Systems, Lecture Notes in Computer Science v. 2491, 2002, p. 240-251.

[10] M. Waldén, K. Sere, Reasoning about Action Systems Using the B method, Formal Methods in System Design, vol.13, 1998, p. 5-35.

[11] http://sourceforge.net/projects/rodin-b-sharp/

[12] C. H. Roth, Digital Systems Design Using VHDL, Belmont, CA USA: CL Engineering, 2007.

[13] IEEE 1076-2008

[14] http://www.altera.com/products/software/sfw-index.jsp

[15] D. Cansell, D. Méry, C. Proch, “System-on-chip design by proof-based refinement”, Springer-Verlag pp. 217-238, March 2009.

[16] J. Plosila, K. Sere, “Action Systems in Pipelined Processor Design”, Proc. 3rd ASYNC Symposium, 1997, pp. 156-166.

[17] T. Seceleanu, Systematic Design of Synchronous Digital Circuits, Turku: TUCS Dissertations, Turku Centre for Computer Science, 2001.

[18] S. Hallerstede, Y. Zimmermann, “Circuit Design by Refinement in EventB”, FDL, pp. 624-637, 2004.

[19] http://www.atelierb.eu/index-en.php

[20] M. Benveniste - A «Correct by Construction» Realistic Digital Circuit – RIAB Workshop – FMWeek 2009 – Eindhoven (http://www.bmethod.com/pdf/riab/st-marc-benvenisteproved-realistic-circuit-handout.pdf), 2009

[21] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge: Cambridge University Press, 1996.

[22] C. Métayer, L. Voisin, The Event-B Mathematical Language, Deploy Eprints, 2007.

[23] M. Leuschel, M. Butler, ProB: A Model Checker for B, Proc. FME, Springer, vol. 2805, 2003, p. 855-874.

[24] http://www.tvset.com.ua/eng/index.php

[25] J.-R. Abrial, Event Model Decomposition, Version 1.3, April 2009.

[26] M. Butler, “Decomposition Structures for Event-B”, in Integrated Formal Methods iFM2009, Springer, 2009, LNCS 5423.

Paper 5

Generation of Structural VHDL Code with Library Components from Formal Event-B Models

Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere

Originally published in:
Proceedings of International Conference on Digital Systems Design (DSD), IEEE Computer Society Conference Publishing Services (CPS), pp. 111-118, 2013.

Extended version published in:
Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Generation of Structural VHDL Code with Library Components from Formal Event-B Models. TUCS Technical Reports 1073, TUCS, 2013.

Extended abstract published in:
Sergey Ostroumov, Leonidas Tsiopoulos, Juha Plosila, Kaisa Sere, Derivation of Structural VHDL from Component-Based Event-B Models. In: Michael Butler, Stefan Hallerstede, Marina Walden (Eds.), Proceedings of the 4th Rodin User and Developer Workshop, TUCS Lecture Notes 18, 31–32, TUCS, 2013.

© 2013 IEEE. Reprinted, with permission, from IEEE.

2013 16th Euromicro Conference on Digital System Design

Generation of Structural VHDL Code with Library Components from Formal Event-B Models

Sergey Ostroumov[1,2], Leonidas Tsiopoulos[2],
Kaisa Sere[2]
[1]TUCS – Turku Centre for Computer Science
[2]Department of IT, Åbo Akademi University
Turku, Finland
{Sergey.Ostroumov, Leonidas.Tsiopoulos,
Kaisa.Sere}@abo.fi

Juha Plosila
Department of Information Technology
University of Turku
Turku, Finland
Juha.Plosila@utu.fi

***Abstract*—We propose a design approach to integrating correct-by-construction formal modeling with hardware implementations in VHDL. Formal modeling is performed within the Event-B framework that supports the refinement approach, i.e., stepwise unfolding of system properties in a correct-by-construction manner. After an implementable model of a hardware system is derived, we apply an additional refinement step in order to introduce hardware library components in the form of functions. We show the mapping between these functions and corresponding library components such that structural, i.e., component-based, VHDL implementation is derived. The application of functions binds unrestricted data types and substitutes regular operations with function calls. The approach is presented through examples that illustrate the additional refinement step and the code generation. We show the advantages in terms of occupied area (2,5% and 12,5%) and performance (13,7% and 15,4%) of the descriptions that incorporate hardware library components.**

Keywords—automated refinement; code generation; design flow; Event-B; formal methods; library components; structural VHDL

I. Introduction

Due to advances in Very-Large-Scale-Integration technology, designers can create increasingly complex systems on a single chip enabling energy-efficient execution of applications. These systems usually consist of a number of components working in unison. However, as complexity of a system grows, it is rather infeasible to perform exhaustive testing in order to guarantee correct behavior of the system.

One of the appropriate approaches for developing correct systems is provided by formal methods. The application of formal methods can be categorized into two techniques. The model-checking [1] technique focuses on extracting a formal model from an implementation and checking some properties on this model. These techniques have been successfully employed (e.g., [2]) to identify errors that were undetected during normal design process. Modification and re-checking of the implementation should then be applied until the required integrity level is achieved.

Another technique to guarantee the correct behavior of a system is offered by a stepwise formal development. The formal modeling is performed following the refinement approach, i.e., unfolding system properties in a correct-by-construction manner. Thus, the formal model of the system is proved correct with respect to its functional requirements introduced as invariants. The utility of this approach can be further enhanced by automated code generation.

For the work in this paper, we utilize the latter approach and use the Event-B formalism [3] as the main framework for formal development. This formalism supports the refinement approach and has adequate tool support – the Rodin platform [4]. This platform is open source software offering the opportunity for an extension of its functionality in the form of plug-ins. Since code generation is a natural step for formal design flow, there are plug-ins that allow one to derive code in software languages such as C, Java, etc. [5]. However, due to the fact that hardware description languages (HDLs) differ in semantics and syntax from software languages, the same methods and techniques cannot be directly and completely applied to hardware design and code generation. Hence, we aim at facilitating the process of HDL descriptions generation from formal models.

The target HDL is the VHSIC Hardware Description Language (VHDL). This language is standardized [6] and widely used in hardware design for systems based on field-programmable-gate-array or application-specific integrated-circuit technologies. VHDL supports the notion of library components allowing the designers to develop a system in a structural, i.e., component-based, manner and to derive possibly optimized code in terms of area and performance.

In this paper, we present a design flow that integrates correct-by-construction formal modeling with hardware implementations in VHDL. The contribution of this paper is a generic approach to deriving component-based formal designs and generating structural VHDL descriptions for them through an additional refinement step. We propose to apply this refinement step to a deterministic implementable model where VHDL library components are introduced into a formal model in the form of functions. We give a subset of components and show the mapping between their formal and informal definitions. The formal library can be further extended with the components used during the design.

To support our approach, we have developed a prototype of a plug-in for the Rodin platform. The plug-in automates the additional refinement step and generates structural VHDL code using components shown in this paper.

`/13 $26.00 © 2013 IEEE
DOI 10.1109/DSD.2013.20

II. Related Work

There exist several formalisms that provide specification and verification of hardware systems such as Signal [7], Esterel [8], ForSyDe [9] and others. Signal is dedicated to data-flow applications domain while Esterel is for control-flow ones. ForSyDe represents the design methodology targeting at covering both domains. The commonality of these languages is that they are all based on the perfect synchrony hypothesis. This hypothesis assumes a zero delay between consuming inputs and producing outputs. In addition, only Signal and ForSyDe support the notion of refinement. Refinement in Signal relies on checking if simulation of inputs and outputs preserves flow-equivalence (model checking) [10]. Refinement in ForSyDe stands for the mapping one process network onto another one restricting these networks to have the same inputs and outputs [9]. Moreover, these transformations have to be performed according to the predefined library.

BlueSpec [11] has been proposed as another solution to formal hardware verification and code generation. The language represents an extension of SystemVerilog and has a sound semantics allowing one to verify certain properties. It also supports design by refinement offering a possibility of integrating automated reasoning into the design flow [12]. However, automated verification of system correctness is provided by external theorem provers and/or model checkers such as PVS [12] and SPIN [22].

Evans [13] describes the mapping of VHDL to B and Communicating Sequential Processes (CSP) methods. The author proposes to derive a B model from VHDL and formalize requirements with CSP. This approach uses a model-checking technique that requires modification and re-checking of the implementation until the desired integrity level is achieved.

In contrast to these approaches, we propose to use the Event-B formalism, which provides data and superposition refinement [14]. These types of refinement allow for stepwise unfolding of system functionality without restricting the model to have the same number of variables in refinements. Furthermore, one can postulate vital properties in terms of invariants for every refinement step. Following this approach, the discharging (proving) proof obligations serves as the guarantee that each refinement step preserves invariants and that concrete refinement step sustains their abstract counterparts. After the required model is derived and proved correct, a structural VHDL description is generated.

Another approach to deriving synchronous hardware systems proposed by Seceleanu [15] relies on Action Systems. The author describes the approach to modeling a synchronous system as read/write operations, where a combinational (asynchronous) circuit that consists of logic gates is followed by a synchronous component, namely a D-flip-flop, which operates on the clock signal. In addition, the author points out the mapping of such modeling to a behavioral VHDL description, where all operations are at one level of code, i.e., the description without components. Despite the fact that the Action Systems framework is similar to the Event-B formalism, it has a different underlying structure, which makes it infeasible to completely apply this approach to Event-B models. Furthermore, in contrast to this approach, we propose to derive component-based models and generate structural VHDL descriptions with library components.

Hallerstede and Zimmermann [16] proposed an approach to VHDL code generation from formal B models. The authors describe the mapping between B models and VHDL code through a middleware language B0, which allows one to generate code without components. This approach is adopted by AtelierB tool and supported by industrial practitioners [17]. Since Event-B is a descendant of B method that allows us to model reactive systems and has a different underlying structure, it is not straightforward how to apply this approach to Event-B models. Furthermore, we consider a component-based design flow, where components are injected into a formal model in the form of functions. This design flow allows for generating a structural VHDL description from such a model.

A similar approach to VHDL code generation has been proposed by Ostroumov and Tsiopoulos [18]. The authors suggest utilizing the conditional statement **if** (condition) **then** action **end if** in the process clause. This guarantees conformance of sequential VHDL behavior to the behavior of its formal counterpart enabling generation of a behavioral (i.e., without components) VHDL description from an implementable model following the usual proof-based design. We adopt and vastly extend the approach of [18]. However, in contrast to this approach, we propose to apply an additional refinement step in order to derive a component-based model and, consequently, a structural VHDL description. The correctness of the additional refinement step is established through the proof obligations of Event-B.

A BHDL tool has been proposed for digital circuit design [23]. The tool converts a VHDL description into B specification with two machines: an abstract that represents a VHDL entity and an implementation that corresponds to architecture. Then, these two machines are verified using the B engine and the VHDL comments are interpreted as invariant properties. In contrast to this approach, we derive an implementable deterministic Event-B model following the usual refinement-based development. Then, components are injected into the model so that a structural VHDL description can be generated.

III. VHDL Description

A. VHSIC Hardware Description Language

VHDL, a standardized hardware description language [6], is widely used in hardware design and is supported by many Computer Aided Design tools (e.g., [20]). A VHDL description consists of two basic elements: an *entity* and an *architecture*. Every entity must have a name and ports. The entity contains two clauses: generic that determines parameters for this entity and port that specifies inputs and outputs of this entity (an interface). The inputs and the outputs are distinguished by the keywords *in* and *out*, respectively.

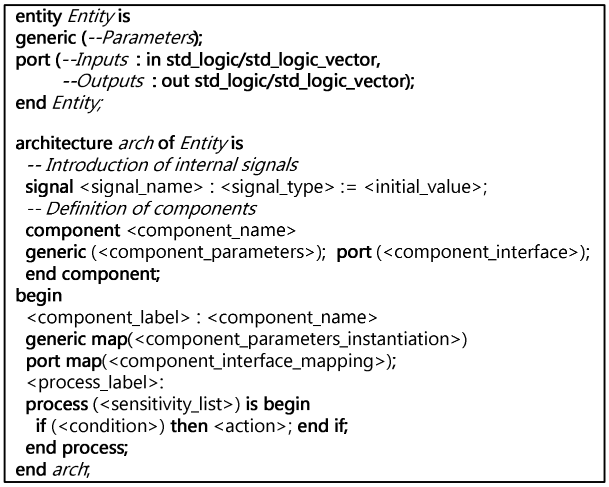
The architecture attached to some entity has a name and a body that describes the behavior (the function) of a hardware component. Inside the architecture, a designer can introduce internal signals and other (e.g., library) components using the keyword component (Fig. 1). A component is simply a predefined entity supplied with an architecture. The component entity has generic parameters that have to be instantiated using the keywords *generic map*. The connection between components is specified by the keywords *port map*. The keywords generic map and port map constitute the architecture body along with the *process* clause. The execution of the process is determined by a so-called *sensitivity list*.

The VHDL action in the process is an assignment to a signal of the form `s <= E`, where `s` is an internal or output signal and `E` is either a constant or an expression over the input and/or internal signals. Every signal whose value is updated has a buffer so that the actual assignment takes place when the whole process completes its execution. Hence, all the signals involved in the process are updated simultaneously.

B. Hardware Library Components

Library components allow the designers to tackle complexity of a system facilitating faster design. Let us review a subset of library components available in Quartus-II software by Altera [20]. A small subset of them is presented in Tab. I, where the components LPM_DIVIDE(DIVIDER) and LPM_DIVIDE(MODULO) differ in the output they produce and the abbreviations ALB, AEB etc. of the LPM_COMPARE component stand for A less than B, A equals to B etc., respectively. However, the library is not limited to the components presented in Tab. I and can be further extended since every library component has a unique definition.

The inputs and the outputs of the library components described here are signals or collections of signals represented by VHDL types STD_LOGIC and STD_LOGIC_VECTOR, respectively. The number of signals of type STD_LOGIC_VECTOR is determined by a constant (a parameter in the generic clause). For the sake of brevity, we exemplify the mapping between a formal model and a structural code by the library component that performs the addition operation (Tab. I, LPM_ADD_SUB(ADDER)). The other components are interpreted in a similar manner.



```
entity Entity is
generic (--Parameters);
port (--Inputs  : in std_logic/std_logic_vector,
        --Outputs  : out std_logic/std_logic_vector);
end Entity;

architecture arch of Entity is
  -- Introduction of internal signals
  signal <signal_name> : <signal_type> := <initial_value>;
  -- Definition of components
  component <component_name>
  generic (<component_parameters>);  port (<component_interface>);
  end component;
begin
  <component_label> : <component_name>
  generic map(<component_parameters_instantiation>)
  port map(<component_interface_mapping>);
  <process_label>:
  process (<sensitivity_list>) is begin
    if (<condition>) then <action>; end if;
  end process;
end arch;
```

Figure 1. VHDL entity and architecture

The component has three parameters: LPM_WIDTH, LPM_DIRECTION and LPM_REPRESENTATION. LPM_WIDTH specifies the number of signals (the width) of the inputs and the output. LPM_DIRECTION determines the type of this component. If it equals to ADD, the component represents an adder. The parameter LPM_REPRESENTATION specifies the type of addition performed (signed or unsigned).

The adder operates on two inputs: the input port DATAA and the input port DATAB. It returns the result of addition of the two inputs to the output port RESULT as well as the carry flag to the output COUT. The input ports and the output port RESULT are of type STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0) while the carry flag is of type STD_LOGIC.

In the next section, we formalize library components as functions within Event-B to achieve correct-by-construction

TABLE I. A SUBSET OF LIBRARY COMPONENTS

Components	Generic	Inputs	Outputs	Operation
LPM_ADD_SUB (ADDER)	LPM_WIDTH, LPM_DIRECTION = "ADD", LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT, COUT	RESULT=(DATAA+DATAB)(LPM_WIDTH-1..0), COUT=(DATAA+DATAB)(LPM_WIDTH)
LPM_ADD_SUB (SUBTRACTOR)	LPM_WIDTH, LPM_DIRECTION = "SUB", LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT	RESULT = (DATAA - DATAB)(LPM_WIDTH-1..0)
LPM_MULT	LPM_WIDTHA, LPM_WIDTHB, LPM_WIDTHP, LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT	RESULT = (DATAA * DATAB)
LPM_DIVIDE (DIVIDER)	LPM_WIDTHN, LPM_WIDTHD, LPM_NREPRESENTATION = "UNSIGNED", LPM_DREPRESENTATION = "UNSIGNED"	NUMER, DENOM	QUOTIENT	QUOTIENT = DATAA ÷ DATAB
LPM_DIVIDE (MODULO)	LPM_WIDTHN, LPM_WIDTHD, LPM_NREPRESENTATION = "UNSIGNED", LPM_DREPRESENTATION = "UNSIGNED"	NUMER, DENOM	REMAIN	REMAIN = DATAA % DATAB

LPM_COMPARE	LPM_WIDTH, LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	AGB,
			AGEB,
			AEB,
			ANEB,
			ALB,
			ALEB
				AGB = bool(DATAA > DATAB),
				AGEB = bool(DATAA ≥ DATAB),
				AEB = bool(DATAA = DATAB),
				ANEB = bool(DATAA ≠ DATAB),
				ALB = bool(DATAA < DATAB),
				ALEB = bool(DATAA ≤ DATAB),

design flow. In addition, we show the correspondence between formal and informal definitions of library components presented in Tab. I.

IV. Event-B Formalization

A. The Event-B Formalism

The Event-B formalism [3] allows designers to develop models in a stepwise and correct-by-construction manner. A specification within Event-B consists of two main elements: a *context* and a *machine*. The context contains static data such as sets, constants, generic theorems and axioms. The machine models the dynamic part, which includes state variables, theorems, invariants specifying system properties that must always hold and events that modify the state variables. The context can be extended by another context and the machine can be refined by another machine. Moreover, the machine can refer to the data defined in a context, if this machine sees this context.

An event within the Event-B framework has the following structure:

$$e \triangleq \textbf{any}\ x\ \textbf{where}\ g\ \textbf{then}\ a\ \textbf{end},$$

where x is a list of local variables, g stands for the guard and a represents an action of the event e, respectively. The guard is a conjunction of predicates that determine the execution of the action. If the guard holds, the action is fired.

The action represents a composition of parallel assignments (denoted as $\|$) that modify state variables. There are three types of assignments in Event-B: deterministic (denoted as :=), non-deterministic from a set (denoted as :∈) and non-deterministic specified by a predicate (denoted as :|).

Each event in Event-B is viewed as a before-after predicate ($BA_e = BA(v, v')$) [3] that links the values of the variables before (v) and after (v') the execution of the event e. This scheme allows us to prove the correctness (consistency) of the model with respect to postulated invariants by discharging proof obligations (POs). In particular, every predicate (i.e., an invariant, a theorem, a guard or an action) has to be well-defined [19], i.e., sound. Each event, in its turn, has to preserve postulated invariants [3, 19]:

$$Inv \wedge g_e \Rightarrow [BA_e]Inv, \qquad \text{(INV)}$$

where Inv is a model invariant whilst g_e and BA_e are the guard and the before-after predicate of the event e, respectively. The expression $[BA_e]Inv$ stands for a substitution in the invariant Inv with the before-after predicate BA_e.

An Event-B model of a system is created in a stepwise manner following the refinement approach. At every refinement step, one adds details towards an implementable model. While refining the model, new variables, invariants, theorems and events can be added. However, the overall behavior of a more concrete model must conform to the overall behavior of its abstraction. This fact is guaranteed through discharging POs guard strengthening (GRD) and action simulation (SIM) [3, 19]:

$$Inv \wedge Inv_r \wedge g_r \Rightarrow g, \qquad \text{(GRD)}$$
$$Inv \wedge Inv_r \wedge BA_{er} \Rightarrow BA_e, \qquad \text{(SIM)}$$

where structures with the sub-script r represent their refined versions.

To ease proving effort when discharging the above POs, one can postulate and prove theorems. Depending on the Event-B element (a context and/or a machine) where a theorem is stated, corresponding POs (THM_c for a context and THM_m for a machine) have to be discharged:

$$A \Rightarrow ThC, \qquad (THM_c)$$
$$A \wedge I \Rightarrow ThM, \qquad (THM_m)$$

where A is a set of axioms defined in a context, I is a set of model invariants, ThC is a theorem postulated in a context whilst ThM is a theorem introduced to a machine.

The Rodin platform [4] supporting the Event-B formalism automatically generates and attempts to discharge the POs stated above. The tool usually achieves high-level of automation (usually over 80%).

B. Event-B Formalization of Library Components

To be able to prove that Event-B formalization conforms to the definitions of hardware library components shown above, we define a function that converts a non-negative decimal number into its binary image. This function binds infinite data types (e.g., naturals) to be suitable for hardware implementation since hardware bit images cannot be infinite.

Definition 1: A bijective function $conv(C, d) = k_b$ converts a non-negative decimal number into its binary image. The parameter $C \in \mathbb{N}1$ determines the upper bound (i.e., the width) on which the function operates. The parameter $d \in 0..2\text{^}C\text{-}1$ represents a non-negative decimal number within the range $0..2\text{^}C\text{-}1$, where $2\text{^}C$ stands for 2 to the power of C. The function returns a binary image of the number d, namely $k_b \in \{x \mid x \in \{0,1\}^* \wedge W(x) = C\}$, where $W(x)$ stands for the number of bits (the width) of the binary number k_b. The function is defined recursively as follows:

$$conv(C,d) = \begin{cases} 0...0_b, \text{ if } d = 0 \\ conv(C,d\text{-}1) +_b 0...1_b, \text{ if } d > 0, \end{cases}$$

where $x...y_b$ is a binary number (e.g., 010_b) whose length (i.e., the number of bits) is determined by the constant C and $n +_b m$ is a binary sum defined as $0_b +_b 0_b = 0_b$, $0_b +_b 1_b = 1_b$, $1_b +_b 0_b = 1_b$, $1_b +_b 1_b = 10_b$.

Example 1. Suppose C equals to 3. Then, any non-negative decimal number from the set $0..2\text{^}3\text{-}1$ (i.e., $0..7$) can be represented as a binary number from 000 to 111:

$conv(3,0) = 000_b$;

$conv(3,5) = conv(3,4) +_b 001_b = conv(3,3) +_b 001_b +_b 001_b = conv(3,2) +_b 001_b +_b 001_b +_b 001_b = conv(3,1) +_b 001_b +_b 001_b +_b 001_b +_b 001_b = conv(3,0) +_b 001_b +_b 001_b +_b 001_b +_b 001_b +_b 001_b = 101_b$.

End of example.

The formalization of library components is performed by using functions applied to an Event-B context. A function f in a context is a constant that has at least two axioms. The first axiom defines the type of the function, i.e., the type of its arguments (T) and returning result (T'):

$$T_1 \times \ldots \times T_n \rightarrow T'_1 \times \ldots \times T'_m,$$

where $T_1 \times \ldots \times T_n$ is the Cartesian product, i.e., the set of all the pairs formed from the types T_1 to T_n.

The second axiom specifies the result returned by the function f:

$$\forall x_i . x_i \in T_i \Rightarrow f(x_1 \mapsto \ldots \mapsto x_n) = Exp(x_1, \ldots, x_n),$$

where i ∈ 1..n and n is the number of arguments that the function f takes (determined by its type). The symbol ↦ represents an ordered pair and allows one to specify a number of arguments for a function. The function f produces the result defined by the expression Exp over x_i.

Following the approach of introducing functions into an Event-B context, we define a formal library of presented hardware components as shown in Tab. II. For instance, let us consider the function add_unsigned in Tab. II that formalizes the VHDL adder component (Tab. I, LPM_ADD_SUB (ADDER)) within Event-B. The type of this function is determined by the first axiom, where add_unsigned_width ∈ ℕ1 is the width. The returning result is specified by the second axiom that models the addition operation of two non-negative numbers.

Theorem (ADD): add_unsigned conforms to LPM_ADD_SUB, where add_unsigned_width = LPM_WIDTH and the parameters LPM_DIRECTION and LPM_REPRESENTATION of LPM_ADD_SUB equal to ADD and UNSIGNED, respectively (ensured by the code generation algorithm described in the next section).

Proof:

1. The function add_unsigned operates on the same input values in decimal as the library component LPM_ADD_SUB in binary:

   $$\forall inp . inp \in 0..(2\text{^}add\_unsigned\_width)\text{-}1 \Rightarrow (\exists inp_b . inp_b = conv(add\_unsigned\_width, inp)),$$

   where inp represents a decimal input to the function while inp_b is a binary image of inp supplied as an input to the component.
2. The result of the function add_unsigned ranges from 0 to 2^(add_unsigned_width+1)-1, i.e., one bit more than the width of the inputs. Hence, the function returns the result as well as the carry flag which corresponds to the value on the outputs RESULT and COUT of the component:

   ∀res . res ∈ 0..2^(add_unsigned_width+1)-1 ⇒
   (∃COUT, RESULT . COUT + RESULT =
   conv(add_unsigned_width+1, res)),

   where res represents the result of the function whereas COUT + RESULT is concatenation of the outputs COUT and RESULT of the component. Clearly, the overflow will never occur.

   Example 2. Suppose add_unsigned_width = LPM_WIDTH = 3, the input ranges of the function and the component are 0..7 and 000..111, respectively, while the result ranges are 0..15 and 0000..1111, respectively. The leftmost (the most significant) bit of LPM_ADD_SUB represents the carry flag.
   End of example.
3. Finally, the definition of the function add_unsigned models addition of two inputs, namely dataa and datab, i.e., the function of the adder component.

Similarly, we can reason about other functions (Tab. II) that specify other library components in Tab. I. □.

TABLE II. COMPONENTS AS EVENT-B FUNCTIONS

Function	Constant(s)	Axioms
add_unsigned	add_unsigned_width	add_unsigned ∈ 0..2^add_unsigned_width-1 × 0..2^add_unsigned_width-1 → 0..2^(add_unsigned_width+1)-1 ∀dataa, datab . dataa ∈ 0..2^add_unsigned_width−1 ∧ datab ∈ 0..2^add_unsigned_width−1 ⇒ add_unsigned(dataa↦datab) = dataa + datab
sub_unsigned	sub_unsigned_width	sub_unsigned ∈ 0..2^sub_unsigned_width−1 × 0..2^sub_unsigned_width−1 → 0..2^sub_unsigned_width−1 ∀dataa, datab . dataa ∈ 0..2^sub_unsigned_width−1 ∧ datab ∈ 0..2^sub_unsigned_width−1 ⇒ (dataa ≥ datab ⇒ sub_unsigned(dataa↦datab) = dataa − datab) ∧ (dataa < datab ⇒ sub_unsigned(dataa↦datab) = 0)
mult_unsigned	mult_unsigned_width_a mult_unsigned_width_b	mult_unsigned ∈ 0..2^mult_unsigned_width_a−1 × 0..2^mult_unsigned_width_b−1 → 0..2^(mult_unsigned_width_a+mult_unsigned_width_b)−1 ∀dataa, datab . dataa ∈ 0..2^mult_unsigned_width_a−1 ∧ datab ∈ 0..2^mult_unsigned_width_b−1 ⇒ mult_unsigned(dataa↦datab) = dataa∗datab
div_unsigned	div_unsigned_width_n div_unsigned_width_d	div_unsigned ∈ 0..2^div_unsigned_width_n−1 × 1..2^div_unsigned_width_d−1 → 0..2^div_unsigned_width_n−1 ∀dataa, datab . dataa ∈ 0..2^div_unsigned_width_n−1 ∧ datab ∈ 1..2^div_unsigned_width_d−1 ⇒ div_unsigned(dataa↦datab) = (dataa ÷ datab)
mod_unsigned	mod_unsigned_width_n mod_unsigned_width_d	mod_unsigned ∈ 0..2^mod_unsigned_width_n−1 × 1..2^mod_unsigned_width_d−1 → 0..2^mod_unsigned_width_d−1 ∀dataa, datab . dataa ∈ 0..2^mod_unsigned_width_n−1 ∧ datab ∈ 1..2^mod_unsigned_width_d−1 ⇒ mod_unsigned(dataa↦datab) = (dataa mod datab)
comp_unsigned	comp_unsigned_width	comp_unsigned ∈ 0..2^comp_unsigned_width−1 × 0..2^comp_unsigned_width−1 → BOOL × BOOL × BOOL × BOOL × BOOL × BOOL ∀dataa,datab . dataa∈0..(2^comp_unsigned_width−1) ∧ datab∈0..(2^comp_unsigned_width−1) ⇒ comp_unsigned(dataa↦datab)= bool(dataa > datab)↦bool(dataa ≥ datab)↦bool(dataa = datab)↦ bool(dataa ≠ datab)↦bool(dataa < datab)↦bool(dataa ≤ datab)

While modeling a system in Event-B, one has to discharge POs (INV), (GRD) and (SIM) to show correctness of the system specification (Section IV). To ease discharging of these POs, we postulated and proved the following theorems (PO (THM_c)). These theorems are available along with the definitions of functions in the library context:

$$\forall n\ .\ n \in \mathbb{N} \Rightarrow 0 < 2\text{^}n, \qquad (ThC_1)$$

$$\forall x,y\ .\ x \in \mathbb{N} \land y \in \mathbb{N} \land x < y \Rightarrow 2\text{^}x < 2\text{^}y, \qquad (ThC_2)$$

$$\forall n\ .\ n \in \mathbb{N} \Rightarrow 2 * 2\text{^}n = 2\text{^}(n+1). \qquad (ThC_3)$$

Theorem (ThC_1) states that 2 to the power of some natural number is a positive number. In other words, the set of values starting from 0 and ending in 2 to the power of some constant is not empty. Hence, the functions formalizing VHDL library components are well-defined on these values. Theorem (ThC_2) shows the order relation between numbers whose powers are in the order relation as well. Theorem (ThC_3) postulates inductiveness of 2 to the power of n.

V. The Design Flow and Code Generation Algorithm

The use of Event-B as a starting point in the design flow of hardware systems facilitates correct-by-construction development with respect to postulated properties and requirements. An automated code generation enhances the utility of the approach reducing testing effort at later design phases. Hence, we propose the design flow shown in Fig. 2, where test cases can be used, e.g., for deploying online testing. The reader is referred to [24] for more details on generation of test cases.

An implementable deterministic model is derived following usual refinement-based development. Then, we apply an additional refinement step that serves as the middleware between a component-based formal model and structural VHDL description. The correctness of this refinement step is established by proving POs (INV), (GRD) and (SIM) using theorems of types (THM_c) and (THM_m) (Section IV). The Rodin platform [4] generates these POs and attempts to prove them automatically. The algorithmic representation of the code generation utilizing the additional refinement step is as follows:

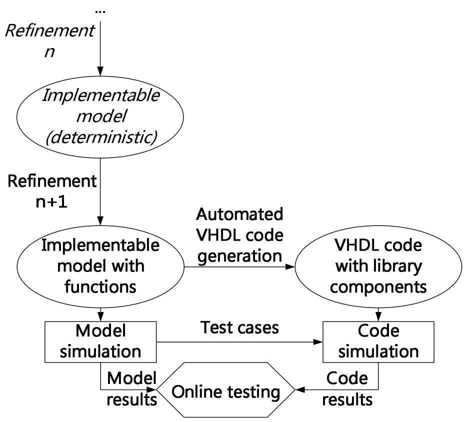

...
Refinement n
Implementable model (deterministic)
Refinement n+1
Automated VHDL code generation
Implementable model with functions
VHDL code with library components
Model simulation
Test cases
Code simulation
Model results
Online testing
Code results

Figure 2. The design flow

1. Refine an implementable model by extending the most definite context (if any) and refining the most concrete machine of the model.
2. Instantiate necessary functions to the newly created context by specifying the set of values they operate with (their width). This set is bounded by the corresponding constants. The necessity of functions is determined by the machine actions.
3. Restrict the types of the state variables according to the specified constants and instantiated functions.
4. Replace regular operations in actions with calls to the corresponding functions.
5. To generate code, interpret each function in the context as a corresponding library component in VHDL according to the defined mapping.
6. Interpret the type of a variable which has been restricted by some constant as STD_LOGIC_VECTOR in terms of VHDL types. The length (the width) number is determined by the corresponding constant.
7. For every component instantiation, introduce an internal VHDL signal connected to the component output(s) in order to allow for chaining of diverse components.

```
invariants
  Voltage_I ∈ ℕ ∧ Current_I ∈ ℕ1 ∧
  Resistance ∈ ℕ ∧ Inputs_Read ∈ BOOL ∧
  (Temp_Read = TRUE ⇒ Resistance = Voltage_I ÷ Current_I) ∧
  // Gluing invariant with a more abstract model
  (Inputs_Read = TRUE ∧ Temp_Read = TRUE ⇒ Temp_I = Resistance)
events
...
event Resist_Comp refines Temp_Read ≙
where  Temp_Read = FALSE ∧ Inputs_Read = TRUE ∧ Current_I ≠ 0
with  Temp = Voltage_I ÷ Current_I
then  Resistance ≔ Voltage_I ÷ Current_I || Temp_Read ≔ TRUE
end
event Compare refines Compare ≙
where  Temp_Read = TRUE ∧ Inputs_Read = TRUE
then Temp_Read ≔ FALSE || Inputs_Read ≔ FALSE ||
  Result_O ≔ bool(Resistance ≥ Temp_Threshold)
end
```

```
invariants
  Voltage_I ∈ 0..2^div_0_unsigned_width_n−1 ∧
  Current_I ∈ 1..2^div_0_unsigned_width_d−1 ∧
  Resistance ∈ 0..2^div_0_unsigned_width_n−1 ∧
  theorem div_0_unsigned(Voltage_I↦Current_I)=Voltage_I÷Current_I
...
events
...
event Resist_Comp refines Temp_Read ≙
where  Temp_Read = FALSE ∧ Inputs_Read = TRUE ∧ Current_I ≠ 0
then  Resistance ≔ div_0_unsigned(Voltage_I↦Current_I) || Temp_Read ≔ TRUE
end
event Compare refines Compare ≙
where  Temp_Read = TRUE ∧ Inputs_Read = TRUE
then  Temp_Read ≔ FALSE || Inputs_Read ≔ FALSE ||
  Result_O :| ∃agb,aeb,aneb,alb,aleb . comp_0_unsigned(Resistance↦Temp_Threshold)=
    agb↦Result_O'↦aeb↦aneb↦alb↦aleb
end
```

Figure 3. Component chaining in separate events

TABLE III. SYNTHESIS RESULTS FOR STATE HOLDING IMPLEMENTATIONS

LE, qt.		**LE, %**	**Tsu, ns**		**Tsu, %**	**Th, ns**		**Th, %**
w/ lib	***w/o lib***		***w/ lib***	***w/o lib***		***w/ lib***	***w/o lib***	
36	37	2,7	9.975	11.562	13,7	2.262	2.215	-2

To support the proposed design flow, we have developed a prototype of the plug-in that automates the additional refinement step and generates structural VHDL description. The plug-in implements the algorithm described above and operates as follows. Firstly, it extends the most definite context of an Event-B project, if any, by copying theorems (ThC_1)-(ThC_3) (Section IV) to it. Secondly, the plug-in traverses the most concrete machine of the project. Each time it sees a regular operation that can be substituted with a function call, the plug-in instantiates a corresponding function available in the library context. A designer specifies the width of the function being instantiated. Thirdly, it refines the most concrete machine and replaces each regular operation with a function call. Fourthly, for every variable involved in such an action, the plug-in generates a type invariant (PO (INV) in Section IV) in order for the types to be feasible for translation. Finally, it applies theorems (ThM) of the form f(x↦y) = x op y to the machine (PO (THM_m) in Section IV), where x and y are the operands and f and op are the function and operation, respectively. For instance, if the function call add_unsigned(x↦y) replaces the expression x+y, then the theorem for this substitution is add_unsigned(x↦y) = x+y. These theorems help in proving the correctness of the additional refinement step.

A specification may contain several identical operations, e.g., two or more addition operations etc. To distinguish them, the plug-in uses an id number that starts from 0 and is increased whenever another function definition is instantiated. For instance, add_0_unsigned, add_1_unsigned, etc. Therefore, each function determines one library component such that the mapping between a formal model and VHDL code is feasible.

VI. EXPERIMENTAL RESULTS

Let us examine a couple of examples showing the application of our method to modeling within the Event-B framework and generating structural VHDL code. The examples show a sequential composition of components using different modeling styles in Event-B.

A. Component Chaining in Separate Events

This example illustrates the use of library components such that the result computed in one event is used as an input for the computations in another event (Fig. 3). Here, we model the calculation of temperature using Ohm's law (event Resist_Comp), where temperature is proportional to resistance (variable Resistance). Then, the obtained value is compared to some threshold and the comparison result is promoted further (event Compare). An instance of this example is aerospace designs domain (e.g., [18, 21]) where the temperature sensor represents a high-quality resistor.

For this model, the Rodin platform generated 57 POs of which 51 were proven automatically. Three POs of type (THM_c) with the proofs were automatically derived for the context theorems (ThC_1)-(ThC_3) (Section IV) by the plug-in. One PO of type (INV) as well as one PO of type (WD) for an automatically generated by the plug-in theorem of type (THM_m) have been proved interactively in a straightforward manner by utilizing theorem (ThC_3). The remaining PO of type (SIM) has then been proved using theorems (THM_m).

We generated VHDL descriptions with and without library components from this model. We then synthesized each description using Quartus-II [20]. The tool analyzed them and provided the information about occupied area and performance. The number of logic elements (LE) measures the area. The worst-case setup time (Tsu) and the worst-case hold time (Th) illustrate the performance of this example. The synthesis results are summarized in Tab. III. They show the advantages in terms of area (2,7%) and performance (13,7%) of the implementation with library components.

B. Replacing Infix Operators with Prefix Function Calls

This example illustrates the model, where a single event produces the result using different operators (Fig. 4). The computation of the result proceeds as follows. The variables Input1_I and Input2_I are multiplied, their result is summed up with the variable Input3_I and, then, this sum is divided by Input1_I. The order in which the operations take place specify the chain of the corresponding hardware library components.

```
invariants
  Input1_I ∈ ℕ1 ∧ Input2_I ∈ ℕ ∧ Input3_I ∈ ℕ ∧
  Result_O ∈ ℕ ∧ Read_Write ∈ BOOL ∧
  Read_Write = FALSE ⇒
    Result_O = (Input1_I∗Input2_I + Input3_I)÷Input1_I
events
…
event Result  ≙
when Read_Write = TRUE
then Read_Write ≔ FALSE ||
    Result_O≔(Input1_I∗Input2_I+Input3_I)÷Input1_I
end
```

```
invariants
  Input1_I ∈ 0..2^mult_0_unsigned_width_a−1 ∧ Input2_I ∈ 0..2^mult_0_unsigned_width_b−1 ∧
  Input3_I ∈ 0..2^add_0_unsigned_width−1 ∧ Result_O ∈ 0..2^div_0_unsigned_width_n−1 ∧
  theorem mult_0_unsigned(Input1_I↦Input2_I)=Input1_I∗Input2_I
…
events
…
event Result refines Result ≙
when Read_Write = TRUE
then Read_Write ≔ FALSE ||
  Result_O≔div_0_unsigned(add_0_unsigned(mult_0_unsigned(Input1_I↦Input2_I)↦Input3_I)↦Input1_I)
end
```

Figure 4. Replacing infix operators with prefix function calls

TABLE IV. SYNTHESIS RESULTS FOR PREFIX FUNCTION CALLS

LE, qt.		**LE, %**	**W-C Tpd, ns**		**W-C Tpd, %**
w/ lib	***w/o lib***		***w/ lib***	***w/o lib***	
28	32	12,5	14,71	17,38	15,4

For this model, the Rodin platform generated 53 POs of which 49 were proven automatically. Three POs (THM_c) with the proofs were automatically copied for the context theorems (ThC_1)-(ThC_3) (Section IV) by the plug-in. The only proof obligation (INV) was proved in an interactive and straightforward manner using theorem (ThC_2).

Analogously to the previous example, we generated VHDL descriptions with and without library components from this model. Then, we used Quartus-II to synthesize each description and acquire information about area and performance. The worst-case time required to propagate the value on the input pin to the output pin (W-C Tpd) reflects the performance metric for this example. Tab. IV summarizes the synthesis results, which show the advantages in terms of area (12,5%) and performance (15,4%) of the description with library components.

VII. CONCLUSION

We have presented a design flow integrating component-based formal modeling within Event-B with structural VHDL implementation. The proposed approach is rather generic allowing one to derive component-based designs in an automated manner. To support the proposed approach, we have developed a prototype of a plug-in that automates the additional refinement step and generation of structural VHDL description. We believe that the application of formal methods at early stages of the design flow with automated code generation can reduce testing effort at later design phases. In addition, we have shown experimental results that illustrate optimization provided by the code with library components (2,5% and 12,5% in area as well as 13,7% and 15,4% in performance). Although the presented experiments are relatively small, the optimization in area and performance of the descriptions with library components is noticeable.

The formal library of hardware components is not limited to the presented components and can clearly be extended. Hence, one future direction is to extend the formal library to facilitate the design of diverse hardware systems as well as to deploy the proposed approach on more complex systems. This may require formalizing complex components using Event-B machines instead of lightweight context functions.

The described components are considered combinational, i.e., clockless. However, there are combinatorial components that depend on the clock signal. Therefore, another direction of our future work is to extend the approach with modeling a system that contains clocked components. This will allow a designer to derive a time-aware model and generate synchronous code from this model.

ACKNOWLEDGMENT

The work is supported by Academy of Finland. Additionally, the authors would like to thank Adjunct Prof. Marina Waldén for the fruitful discussions and valuable feedback.

REFERENCES

[1] E. Clarke, Model Checking, Cambridge: The MIT Press, 2002.

[2] A. Roychoudhury, T. Mitra, S.R. Karri, Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol, Design, Automation and Test in Europe conference (DATE), IEEE, pp. 828-833, 2003.

[3] J.-R. Abrial, Modeling in Event-B. System and Software Engineering, Cambridge: Cambridge University Press, 2010.

[4] The Rodin platform. Available: http://www.event-b.org/install.html

[5] S. Wright, Automatic Generation of C from Event-B, Workshop on Integration of Model-based Formal Methods and Tools, p. 14, 2009.

[6] IEEE Standard VHDL Language Reference Manual, IEEE 1076, 2008.

[7] A. Benveniste, P. Le Guernic, Hybrid Dynamical Systems Theory and the Signal Language, IEEE Transactions on Automatic Control 35(5), pp. 535-546, 1990.

[8] D. Potop-Butucaru, R. de Simone, Optimizations For Faster Execution Of Esterel programs, Proc. of MEMOCODE conference, 2003, pp. 227-236.

[9] I. Sander, A. Jantsch, System Modelling and Transformational Design Refinement in ForSyDe, Transactions on Computer Aided Design of Integrated Circuits and Systems, IEEE, Vol. 23, 2004, pp. 17-32.

[10] J. Talpin, P. Guernic, S. Shukla, R. Gupta, F. Doucet, Polychrony for Formal Refinement Checking in a System-Level Design Methodology, Application of Concurrency to System Design (ACSD), IEEE, pp. 9-19, 2003.

[11] BlueSpec Documentation. Available: http://www.ece.ucsb.edu/its/bluespec/index.html.

[12] D. Richards, D. Lester, A monadic approach to automated reasoning for BlueSpec SystemVerilog, Innovations System Software Engineering, Springer-Verlag, pp. 85-95, 2011.

[13] N. Evans, Integrating Formal Methods with Informal Digital Hardware Development, Proc. of AVoCS, 2010, p. 16.

[14] R. J. R. Back, K. Sere, "Superposition Refinement of Reactive Systems", Formal Aspects of Computing, Springer, Vol. 8, 1995, pp. 324-346.

[15] T. Seceleanu, Systematic Design of Synchronous Digital Circuits, Turku: TUCS Dissertations, Turku Centre for Computer Science, 2001.

[16] S. Hallerstede, Y. Zimmermann, "Circuit Design by Refinement in Event-B", FDL, pp. 624-637, 2004.

[17] M. Benveniste, A «Correct by Construction» Realistic Digital Circuit, RIAB Workshop, FMWeek, 2009.

[18] S. Ostroumov, L. Tsiopoulos, VHDL Code Generation from Formal Event-B Models. IEEE Digital System Design, 14th Euromicro Conference, Oulu, 2011, pp. 127-134.

[19] K. Robinson. (2011, June, 28). System Modelling & Designing using Event-B. Available: http://www.cse.unsw.edu.au/~cs9116/PDF/SMD.pdf

[20] Quartus-II software. Avaliable: http://www.altera.com/products/software/sfw-index.jsp

[21] B. Nuckolls, Practical Low Resistance Measurements, 2004. Available: www.aeroelectric.com/articles/LowOhmsAdapter_3.pdf

[22] G. Singh, E. Shukla, Verifying Compiler bsed Refinement of Bluespec Specifications using the SPIN model Checker, 15th International SPIN Workshop, Springer, pp. 250-269, 2008.

[23] A. Aljer, P. Devienne, S. Tison, BHDL: Circuit design in B, Conference on Application of Concurrency to System Design, IEEE, pp. 1-2, 2003.

[24] S. Ostroumov, L. Tsiopoulos, J. Plosila, K. Sere, Generation of Structural VHDL Code from Formal Event-B Models, TUCS Report 1073, 2013.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-3219-0
ISSN 1239-1883

Sergey Ostroumov

Agent-Based Management Systems for Many-Core Platforms

Sergey Ostroumov

Agent-Based Management Systems for Many-Core Platforms