TUCS

Fredrik Degerlund

Scheduling of Guarded Command Based Models

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Dissertations
No 152, December 2012

Scheduling of Guarded Command Based Models

Fredrik Degerlund

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma on December 10, 2012, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahainengatan 3-5, FIN-20520 Åbo, Finland

2012

Supervisors

Prof. Kaisa Sere
Department of Information Technologies
Åbo Akademi University
Joukahainengatan 3-5, FIN-20520 Åbo
Finland

Docent Marina Waldén
Department of Information Technologies
Åbo Akademi University
Joukahainengatan 3-5, FIN-20520 Åbo
Finland

Reviewers

Prof. Einar Broch Johnsen
Department of Informatics
University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo
Norway

Dr. Thai Son Hoang
Institute of Information Security
Department of Computer Science
ETH Zentrum, CH-8092 Zürich
Switzerland

Opponent

Prof. Einar Broch Johnsen
Department of Informatics
University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo
Norway

ISBN 978-952-12-2811-7
ISSN 1239-1883

Abstract

Formal methods provide a means of reasoning about computer programs in order to prove correctness criteria. One subtype of formal methods is based on the weakest precondition predicate transformer semantics and uses guarded commands as the basic modelling construct. Examples of such formalisms are Action Systems and Event-B. Guarded commands can intuitively be understood as actions that may be triggered when an associated guard condition holds. Guarded commands whose guards hold are non-deterministically chosen for execution, but no further control flow is present by default. Such a modelling approach is convenient for proving correctness, and the Refinement Calculus allows for a stepwise development method. It also has a parallel interpretation facilitating development of concurrent software, and it is suitable for describing event-driven scenarios. However, for many application areas, the execution paradigm traditionally used comprises more explicit control flow, which constitutes an obstacle for using the above mentioned formal methods. In this thesis, we study how guarded command based modelling approaches can be conveniently and efficiently scheduled in different scenarios. We first focus on the modelling of trust for transactions in a social networking setting. Due to the event-based nature of the scenario, the use of guarded commands turns out to be relatively straightforward. We continue by studying modelling of concurrent software, with particular focus on compute-intensive scenarios. We go from theoretical considerations to the feasibility of implementation by evaluating the performance and scalability of executing a case study model in parallel using automatic scheduling performed by a dedicated scheduler. Finally, we propose a more explicit and non-centralised approach in which the flow of each task is controlled by a schedule of its own. The schedules are expressed in a dedicated scheduling language, and patterns assist the developer in proving correctness of the scheduled model with respect to the original one.

Sammanfattning

Formella metoder erbjuder ett sätt att resonera om datorprogram med målsättningen att bevisa korrekthetskriterier. En underkategori av formella metoder baserar sig på predikattransformerarsemantiken för svagaste förvillkor och använder vaktade kommandon som grundläggande modelleringskonstruktion. Aktionssystem och Event-B utgör exempel på sådana formalismer. Vaktade kommandon kan intuitivt uppfattas som aktioner förknippade med ett vaktvillkor, och som kan utlösas under förutsättning att villkoret uppfylls. Vaktade kommandon vilkas villkor är uppfyllt väljs ut icke-deterministiskt för att exekveras, men utöver det finns det i standardutförandet ingen ytterligare flödeskontroll. Ett sådant modelleringstillvägagångssätt är behändigt för att bevisa korrekthet, och preciseringskalkylen erbjuder en metod för stegvis utveckling. Metoden omfattar också ett sätt att tolka modellerna i termer av samtidig exekvering, vilket underlättar utveckling av parallella program, och den är också lämplig för att beskriva händelsedrivna scenarier. I många tillämpningsområden används emellertid traditionellt en exekveringsparadigm som omfattar mer explicit flödeskontroll, vilket försvårar användandet av ovannämnda formella metoder. I denna avhandling undersöker vi hur modelleringstekniker baserade på vaktade kommandon kan användas i olika scenarier på ett effektivt och behändigt sätt. Vi fokuserar först på modellering av social pålitlighet för transaktioner i miljön av ett socialt nätverk. På grund av scenariots händelsebaserade natur visar det sig vara relativt enkelt att använda vaktade kommandon. Vi fortsätter genom att undersöka modellering av parallella program, med särskild tonvikt på beräkningstunga scenarier. Vi går från teoretiska överväganden till att undersöka genomförbarheten av praktiska implementationer genom att utvärdera prestandan och skalbarheten i en fallstudie. Denna studie utgörs av parallell exekvering via automatisk schemaläggning utförd av en dedicerad schemaläggare. Slutligen föreslår vi ett mera explicit icke-centraliserat tillvägagångssätt där flödet av varje process kontrolleras av ett eget schema. Schemorna uttrycks i ett dedicerat schemaläggningsspråk, och utvecklaren kan använda mönster för att bevisa korrektheten i den schemalagda modellen med avseende på funktionaliteten av den ursprungliga versionen.

Acknowledgements

Pursuing a PhD degree is a major undertaking. While the book you are currently reading constitutes my personal thesis, it would not have been possible without the direct and indirect involvement and support of many other people.

First of all, I want to thank my supervisors Prof. *Kaisa Sere* and Docent *Marina Waldén* for their valuable support, not only in theoretical matters regarding refinement calculus, action systems and Event-B, but also for their practical guidance throughout my PhD studies. My supervisors, as well as Dr. *Pontus Boström*, have also played a crucial role in co-authoring a number of original publications that make up a part of my thesis. I have also had many informal luncheon discussions on formal methods with Pontus. Furthermore, I want to direct my thanks to Prof. *Einar Broch Johnsen* for acting as my opponent during the defence. He, as well as Dr. *Thai Son Hoang*, also kindly agreed to review my thesis. I am grateful to both of them for taking the time to read, evaluate and give important feedback on my work.

Mats Neovius and I have frequently knocked on each others' doors for constructive discussions on refinement calculus, action systems, social trust, or just about anything that PhD students talk about. Working together with *Richard Grönblom* was a positive experience, and his programming related to code generation and scheduling turned out to be very valuable for my research. Conversations with *Petter Sandvik* have traditionally taken place at luncheon time, and we have covered all kinds of topics from technology to soft drinks. *Johannes Eriksson* has every now and then been spotted in the coffee room or in the hallways, and such encounters have many a time resulted in spontaneous discussions. The positive attitude of *Linas Laibinis* and the jokes of *Ragnar Wikman* have very much contributed to the atmosphere at the department, and I would also like to express my appreciation to Prof. *Ralph-Johan Back* for his work on the Refinement Calculus. Many other researchers, teachers and students have also played an important role at the department during my studies, including, but not limited to, *Luigia Petre*, *Elena Troubitsyna*, *Marta Olszewska*, *Leonidas Tsiopoulos*, *Anton Tarasyuk*, *Torbjörn Lundkvist*, *Kim Solin*, *Qaisar Malik* and *Larissa Meinicke*.

I am also very grateful towards the administrative and technical staff both at the department and at TUCS. *Christel Engblom* has been around to handle a wide range of practical arrangements, while *Britt-Marie Villstrand*, *Irmeli Laine*, *Tomi Mäntylä* and many more have taken care of other administrative tasks. *Magnus Dahlvik* has repeatedly helped me out in all kinds of matters regarding my work computer.

My PhD studies have primarily been financed by TUCS and by the European Union project DEPLOY, as well as the EFFIMA program coordinated by FIMECC. It goes without saying that I want to express me deepest gratitude to them for making it possible to pursue this project in the first place. I also want to direct my thanks to Hans Bang Stiftelsen, Stiftelsen för Åbo Akademi and all other organisations that have supported my research financially.

Last, but not least, I want to thank all of my family and friends. My parents *Kaj Degerlund* and *Gunhild Degerlund* have been around to support me throughout the entire project. I also want to mention my grandmother *Hertta Lundström* and my late grandfather *Herbert Lundström*. My friends have given me the opportunity to also focus on other aspects of life, and I have definitely enjoyed all the spelkvällar I have had with *Johan Hindström* and *Marina Asplund* throughout all these years.

Åbo, November 2012
Fredrik Degerlund

List of Original Publications

I. F. Degerlund and K. Sere. “A Framework for Incorporating Trust into Formal Systems Development”. In: *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Proceedings*. Ed. by Z. Liu C. B. Jones and J. Woodcock. Vol. 4711. Lecture Notes in Computer Science. Springer, 2007, pp. 154–168.

II. F. Degerlund and K. Sere. “Refinement of Parallel Algorithms”. In: *Process Algebra for Parallel and Distributed Processing*. Ed. by M. Alexander and W. Gardner. Computational Science Series. Chapman & Hall / CRC Press (Taylor & Francis Group), 2008, pp. 77–96.

III. F. Degerlund, K. Sere and M. Waldén. “Implementation Issues Concerning the Action Systems Formalism”. In: *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT’07)*. Ed. by D. S. Munro, H. Shen, Q. Z. Sheng, H. Detmold, K. E. Falkner, C. Izu, P. D. Coddington, B. Alexander and S.-Q. Zheng. IEEE Computer Society Press, 2007, pp. 471–479.

IV. F. Degerlund. *Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B*. Tech. rep. 1051. Turku Centre for Computer Science (TUCS), 2012.

    Based on [30]: F. Degerlund. “Scheduling of Compute-Intensive Code Generated from Event-B Models: An Empirical Efficiency Study”. In: *Proceedings of Distributed Applications and Interoperable Systems (DAIS) 2012*. Ed. by K. Göschka and S. Haridi. Vol. 7272. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 177–184.

V. P. Boström, F. Degerlund, K. Sere and M. Waldén. “Derivation of concurrent programs by stepwise scheduling of Event-B models”. In: *Formal Aspects of Computing* (2012). DOI: 10.1007/s00165-012-0260-5.

Contents

Part I

Research Summary

Chapter 1

Introduction

Software testing [64] has often been the method of choice in the quest to enforce software correctness. However, as pointed out by Dijkstra [24], testing can show the presence of bugs, but not their absence. History has shown that even highly funded major projects are at risk of software errors. For example, in 1996, a software controller error caused the Ariane 5 rocket of Flight 501 to veer off its flight path and explode [54]. While the rocket was unmanned and there were no human casualties, the incident resulted in considerable economic and scientific losses. In contrast, there were several deaths and serious injuries resulting from erroneous software in the Therac-25 radiation therapy machine [53]. In the six known incidents, which took place from June 1985 to January 1987, the patients received massive radiation overdoses.

1.1 Informal and Formal Methods

There are many different software development processes that can improve the quality of the resulting software. A traditional approach is the waterfall model [67], containing sequential steps such as requirements specification, design, implementation, testing and maintenance. More recently, agile development [17] methods have been proposed, in which the focus is shifted from processes to individuals, from following plans to responding to changes, etc. Such an agile method is Extreme Programming [16], which involves unit testing and focuses on pair programming as a mechanism of mitigating low quality code.

The above mentioned development processes are informal in the sense that they do not rely on strict mathematical-logical reasoning. This can be contrasted to formal methods, which allow for rigorous reasoning about software and proving properties thereof. Such properties typically constitute some form of correctness criteria. Important mathematical-logical foundations for reasoning about computer programs were laid in the 1960s [40, 45], and formal methods provide a potentially powerful means of improving software quality. The methods and tools have evolved over time, and a study of their industrial applications [18] shows that many application areas have benefited from formal modelling and verification, yet many challenges remain. The current state of formal methods has also been questioned elsewhere [65]. This calls for more research in the field.

1.2 Classification of Formal Methods

Formal methods do not constitute a uniform class of techniques. The mathematical-logical foundations may differ considerably from method to method, but they can roughly be classified into three categories:

1. Model checking methods
2. State-based methods
3. Process algebras

In model checking [27, 66], the basic idea is to verify that certain properties hold for a given system model. The properties are expressed in the form of a specification in a temporal logic (e.g. LTL or CTL). A challenge is the state space explosion problem, whereby the number of states to consider grows exponentially. To mitigate this phenomenon, state reduction techniques such as bisimulation can be used.

The state-based methods focus on state space transitions of a given model, brought about by the execution of commands. In this way, model consistency can be checked, e.g. with respect to an invariant. Moreover, refinement [6, 7, 13, 14, 35, 62, 63, 73] is typically also supported. Refinement makes it possible to gradually turn an abstract specification into more and more concrete ones through the means of correctness preserving transformations. The Guarded Command Language [36, 37], the Action Systems formalism [8, 9], the B-method [2], Event-B [1, 39], the Z notation [22, 70], the Vienna Development Method (VDM) [19] and UNITY [26] all constitute examples of state-based formalisms.

Finally, the process algebras, or process calculi, deal with communication between processes. They are also known as event-based formalisms, where the word event refers to the interactions taking place between the processes. Notable examples of process algebras are CSP [46], CCS [60] and the pi-calculus [61].

It should be noted that the above classification is only indicative. For example, Circus [25] is a hybrid approach combining Z, which is state-based, with the process algebra CSP. Furthermore, the Creol language [29, 50] is inspired by process algebras, but differs from most of them in using asynchronous rather than synchronous communication [49]. Creol is also object-oriented and operates on a higher abstraction level than traditional process algebras, and, in fact, than most formal methods in general. There are also light-weight formalisms which combine informal and formal elements. Such an approach is the Eiffel language [59], which inherently supports *Design by Contract*. This allows the programmer to take advantage of concepts such as *preconditions*, *postconditions* and *invariants*, which are traditionally not found in programming languages.

1.3 Guarded Commands and Scheduling

This thesis is not about program correctness in general, nor about formal methods as a whole. Instead, we focus on a specific class of modelling languages within the category of state-based formal methods. In the previous section, we mentioned the Guarded Command Language as an example of a state-based formalism. The language, proposed by Dijkstra together with the *weakest precondition* predicate transformer semantics, provides a basic but elegant means of reasoning about programs.

The language introduces the concept of *guarded commands*, a simplified form of which can be constructed as follows:

$$\langle \textit{guarded command} \rangle ::= \langle \textit{guard} \rangle \rightarrow \langle \textit{statement} \rangle \tag{1.1}$$

The *guard* is a predicate on the state of the model, whereas the *statement* is a command that updates the state. The intuitive interpretation of a guarded command is that the statement is eligible for execution only when the guard holds, i.e. evaluates to true. The guarded command is then said to be *enabled* (the opposite being *disabled*). Guarded commands are typically combined using the *non-deterministic demonic choice* operator $[\!]$ and included in a **do od** loop, resulting in the following construct:

```
do
    $g_1 \rightarrow S_1$
  $[\!]$ $g_2 \rightarrow S_2$
  ...
  $[\!]$ $g_n \rightarrow S_n$
od
```

This can be understood so that any one of the n guarded commands is non-deterministically chosen for execution, under the condition that it is enabled at the time of choice. Because of the loop, this procedure is repeated until no guarded command is enabled any longer. Note that the choice is *demonic*, which means that the model designer is not in control of the choice. While these interpretations are informal, the precise semantics will be given in the next chapter in terms of weakest precondition predicate transformers. We will also mention how weakest preconditions can be used to form a correctness preserving refinement chain, making it possible to develop programs in a stepwise manner.

The principles of the guarded commands have been incorporated into a number of newer specification languages, such as the Action Systems formalism and Event-B. We will refer to such languages as *guarded command based languages*, and they constitute the object of study in this thesis. More specifically, we are interested in how the theoretical concept of demonic non-deterministic choice is controlled and implemented in practice. Such *scheduling* aspects are important from a practical point of view, and, by extension, industrial adoption. Guarded command based languages are particularly suitable for modelling parallel software, and, therefore, we focus especially on parallel and ubiquitous scenarios.

1.4 Structure of the Thesis

The thesis is based on five original papers. Reprints of these articles are provided in Part II. The purpose of Part I is to provide background information, to summarise the contributions of the publications, as well as to provide further discussion and analysis. The text has deliberately been written as a high level overview, and we refer to the the original papers for more technical details.

The rest of Part I is structured as follows. In Chapter 2, we provide background information on guarded commands and languages based on them. We also discuss associated scheduling issues that we attempt to address in the thesis. In Chapter 3, we present the original publications, focusing on their respective contributions in context of the topic of the thesis. The purpose of Chapter 4 is to further analyse and discuss the approaches taken in the papers. We also compare the methods to each others as well as to related work, and discuss their respective merits and drawbacks when applied in different scenarios. Finally, in Chapter 5, we give a short summary of what we have analysed and achieved in the thesis.

Chapter 2

Background and Motivation

In this chapter, we give a short theoretical background to our field of study. We first deal with predicates and weakest precondition predicate transformers. These fundamentals underpin the refinement relation, which, in turn, plays an important role in guarded command based languages. After providing an introduction to this category of languages, we move on to discuss scheduling and explain why it is motivated to study it. We also give a problem statement as well as short summary of the research methods used in the thesis. Much of the theoretical background in this chapter is based on the work of Back and von Wright [13].

2.1 Weakest Precondition Predicate Transformers

In Chapter 1, we mentioned that guarded commands can be given a formal semantics in terms of weakest precondition predicate transformers [13, 36]. Since guarded command based languages are state-based, focus is on the state of a model as well as updates thereof. All possible configurations of the variables in a model make up the *state space*, for which we use the denotation Σ.

Before going into the details of predicate transformers, we shortly discuss the underlying concept of *predicates*. A predicate q on Σ is a function $q : \Sigma \to \mathsf{BOOL}$ mapping states to Boolean values in $\mathsf{BOOL} = \{\mathsf{F}, \mathsf{T}\}$. Two important special cases of predicates are the constant functions false and true, which map any state to the Boolean values F and T, respectively. They are, thus, defined as $\mathsf{false}(s) = \mathsf{F}$ and $\mathsf{true}(s) = \mathsf{T}$ for any state s in Σ. Predicates form a *lattice* with false as the bottom element, true as the top element, and implication ($\Rightarrow$) as the ordering relation [13]. It is sometimes convenient to think of predicates in terms of subsets of the state space, instead of the function as such. The subset then consists of all such states s in Σ for which the predicate function evaluates to T.

Predicate transformers can now be defined as functions mapping one predicate to another, i.e. they have the type $(\Sigma \to \mathsf{BOOL}) \to (\Sigma \to \mathsf{BOOL})$. The idea is to use them as a means of giving a semantics for *statements* (commands) in the modelling language. More specifically, a weakest precondition predicate transformer takes a postcondition predicate q as input, and returns a predicate reflecting the *weakest* predicate (smallest w.r.t. the implication ordering) from which execution of a statement S terminates in a state where q holds. The semantics of individual statements is, thus, defined in terms of their associated predicate transformers. The following list gives the weakest precondition predicate transformers associated with different statements:

$(v := E)(q)$	$=$	$q[v := E]$	((Multiple) assignment)
$(S;T)(q)$	$=$	$S(T(q))$	(Sequential composition)
$(S \mathbin{[\!]} T)(q)$	$=$	$S(q) \wedge T(q)$	(Demonic non-determ. choice)
$S^{\omega}(q)$	$=$	$\mu X.(S; X \mathbin{[\!]} \mathsf{skip})(q)$	(Strong iteration)
$S^{*}(q)$	$=$	$\nu X.(S; X \mathbin{[\!]} \mathsf{skip})(q)$	(Weak iteration)
$\mathsf{skip}(q)$	$=$	q	(Stuttering)
$\mathsf{magic}(q)$	$=$	true	(Miraculous behaviour)
$\mathsf{abort}(q)$	$=$	false	(Aborting behaviour)
$[p](q)$	$=$	$p \Rightarrow q$	(Assumption)
$\{p\}(q)$	$=$	$p \wedge q$	(Assertion)

Here, v $(v_1, \ldots, v_m)$ is a list of variables, E $(E_1, \ldots, E_m)$ is a list of expressions, p and q represent predicates, and S as well as T are arbitrary statements. Moreover, $q[v := E]$ refers to textual substitution, i.e. all free occurrences of v_i in q are replaced by E_i for all $i \in \{1, \ldots, m\}$. The notations μ and ν represent *least fixed point* and *greatest fixed point*, respectively.

The statement $v := E$ is a (multiple) assignment, where the variables of v are assigned the corresponding expression in E. Sequential composition (;) composes two arbitrary statements for sequential execution, whereas demonic non-deterministic choice ($[\!]$) means that either one of the statements will be executed. However, the choice is not up to the modeller, hence the name “demonic”. Strong iteration, intuitively, repeatedly executes a given statement a demonically chosen and possibly infinite number of times. Weak iteration is similar, but the number of executions is finite. The skip statement represents *stuttering*, i.e. it will leave the state unchanged. The statement magic models miraculous behaviour, which means that any postcondition is achieved regardless of the state in which it is executed. This is in contrast to abort, which does not guarantee to achieve any postcondition at all. It does not necessarily even terminate. Finally, we have assumption $[p]$ and assertion $\{p\}$. If executed in a state where p holds, they both behave as skip. However, if p does not hold in the pre-state, assumption behaves as magic, whereas assertion behaves as abort.

Similarly to the predicates, the statements also form a lattice, provided that they are monotonic [63]. All statements we consider here fulfil the monotonicity criterion. The bottom element is false, whereas the top element is true. The statement lattice is ordered under the *refinement* relation, which we will discuss later.

The predicate transformers associated with statements provide us with a formal semantics for mathematical-logical reasoning. The ultimate goal is typically to prove correctness in one form or another. A well-established notion of correctness is that of *Hoare triples* [45], which are constructs consisting of a precondition p, a postcondition q and a statement S:

$$\{p\}\ S\ \{q\} \tag{2.1}$$

The meaning of such a Hoare triple is that if statement S is executed in a state where p holds, then it is guaranteed to terminate in a state where q holds. Despite the similarities in notation, the curly brackets of Hoare triples are not to be confused with assertion statements. Also note that we here deal with *total correctness*, which means that we do not consider the triple to hold if the statement is non-terminating. This is in contrast to the slightly different notion of *partial correctness*, according to which a triple is said to hold if it either terminates and achieves q or it does not terminate at all. It is worth noting that statements can be linked to total correctness Hoare triples through the means of their respective predicate transformers:

$$S(q) = \{p \mid \{p\}\ S\ \{q\}\} \tag{2.2}$$

This important connection demonstrates why predicate transformers have the potential to be useful for reasoning about the correctness of programs.

2.2 Refinement

Sometimes it is not enough to consider the correctness of a single statement, but we may be interested in how two statements are related to each other in terms of correctness. Such reasoning can be achieved through the means of *refinement*, which is particularly useful for stepwise development of programs. Refinement has been extensively studied in the literature [6, 7, 13, 14, 35, 62, 63, 73], and it can be defined as follows. Consider two statements S_1 and S_2. Statement S_2 is then said to be a refinement of S_1, denoted by $S_1 \sqsubseteq S_2$, if:

$$\forall q.\ S_1(q) \Rightarrow S_2(q) \tag{2.3}$$

Stating that $S_1 \sqsubseteq S_2$ can also equivalently be formulated in terms of total correctness Hoare triples as follows:

$$\forall p, q.\ \{p\}\ S_1\ \{q\} \Rightarrow \{p\}\ S_2\ \{q\} \tag{2.4}$$

This intuitively means that whatever postcondition S_1 is able to achieve, S_2 will also be able to achieve it. Statement S_2 may or may not be able to achieve a stronger postcondition than S_1, but the important aspect is that it will not be weaker. Because of this, the refined statement S_2 can be said to preserve correctness w.r.t. the original statement S_1. The refinement is said to be more *concrete* than the statement it refines, which, in turn, is more *abstract* than the refinement. Note that the refinement relation is *reflexive* and *transitive*, which means that it is possible to create chains of refinements, whereby the most concrete statement is also necessarily a refinement of the most abstract one.

Since relatively complex statements can be constructed from more basic ones using iteration, choice and sequential composition, statements can be used to model specifications/programs. Starting from an initial specification, more concrete specifications can be be derived and shown to be refinements of the initial one. This stepwise refinement approach, which is illustrated in Figure 2.1, allows for a formal and correctness-preserving means of developing software. Since correctness is an integral part of the development process, the term *correct by construction* can be used. Note that no clear distinction between specifications and (executable) programs has to be made. In fact, an important goal of the Refinement Calculus [7, 13, 14] was to eliminate previous restrictions requiring executability of statements [36]. However, from a practical point of view, it is important that the final version of a program only contains executable statements.

Example 1 Consider a variable x and two integers i_1 and i_2. The statement $x := i_1 \mathbin{[\!]} x := i_2$ is then refined by $x := i_2$, i.e.:

$$x := i_1 \mathbin{[\!]} x := i_2 \sqsubseteq x := i_2 \tag{2.5}$$

That this is the case can be verified by resorting to the definition of refinement as well as the predicate transformers of the two statements. This gives us the following expression, which is obviously true:

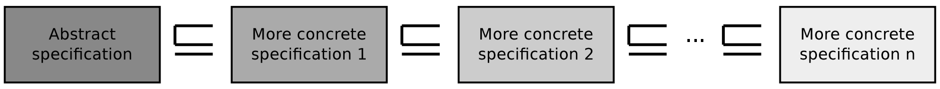
$$\forall q.\ q[x := i_1] \land q[x := i_2] \Rightarrow q[x := i_2] \tag{2.6}$$

This constitutes an example of how non-deterministic choice can be reduced through the means of refinement, going from abstract statements towards more concrete ones.

Example 2 The following refinement chain holds:

$$\mathsf{abort} \sqsubseteq x := i_1 \sqsubseteq [x = i_2]; x := i_1 \sqsubseteq \mathsf{magic} \tag{2.7}$$

This can also be easily verified using the definition of refinement. The fact that the other statements (in fact, *any* other statements) are refinements of



Abstract specification
More concrete specification 1
More concrete specification 2
More concrete specification n

Figure 2.1: Refinement chain illustrating stepwise development of programs.

abort can also be deduced from the fact that abort constitutes the bottom element of the statement lattice. Similarly, magic, being the top element, is a refinement of the other statements. The statement $[x = i_2]; x := i_1$ has the same effect as $x := i_1$ if $x = i_2$ initially holds, or else it behaves like magic (miraculous behaviour). This means that it is a more concrete statement than $x := i_1$ in the Refinement Calculus sense of the word. This refinement chain also shows us that more concrete behaviour does not in every single case correspond to practical implementability. In fact, magic cannot be implemented, and statements can from this point of view be refined "too far".

2.3 Guarded Command Based Languages

We shortly presented the Guarded Command Language in Chapter 1, and we will now discuss guarded command based languages in light of the theory in the above sections. However, we have not introduced any statements for guarded command, nor for the **do od** loop. These can, however, be defined in terms of the statements above as follows:

$$g \to S = [g]; S \tag{2.8}$$

$$\textbf{do } B \textbf{ od} = B^\omega; [\neg gd(B)] \tag{2.9}$$

It is common (but not mandatory) to require that S is *strict with respect to* false, i.e. that $S(\mathsf{false}) = \mathsf{false}$. This guarantees that the the "guard portion" of $g \to S$ is indeed confined to g. As for the **do od** loop, B is a statement representing the loop body. Furthermore, $gd(B)$ stands for $\neg B(\mathsf{false})$, which represents the states from which execution of B will not behave as magic (miraculous behaviour), i.e. it extracts the guard portion of the statement. The assumption prevents termination of the loop until the guard portion of B is false.

Dijkstra considered guarded commands to be of a distinct syntactic category than statements [36]. This is in contrast to the definition above, where the whole guarded command $g \to S$ is seen as a statement, and not only the non-guard part S. Treating guarded commands as statements is possible because of allowing for assumptions and miraculous behaviour, which was not supported by Dijkstra[1]. The **do od** loop is also defined differently

[1]The *law of the excluded miracle* was lifted in the Refinement Calculus.

than in Dijkstra's work. The definition used here is convenient for algebraic reasoning about loops [12].

2.3.1 Action Systems

We now give a schematic view of a model in the Action Systems formalism [8, 9]:

```
$\mathcal{A} = |[$ var $v_1, v_2, ..., v_m$
        $S_0$;
        do
              $g_1 \rightarrow S_1$
           $[]\ g_2 \rightarrow S_2$
           $\ldots$
           $[]\ g_n \rightarrow S_n$
        od
     $]|$
```

The **var** clause contains a list of variables, S_0 is a statement initialising the state, and it is sequentially followed by a **do od** loop, similar to the one in the Guarded Command Language as presented in Chapter 1. An action system, as defined above, can be seen as a statement in its own right, i.e. as a weakest precondition predicate transformer. This implies that it can also be refined accordingly, giving us a means of stepwise refinement of action systems. This approach assumes that we are only interested in *total correctness*, i.e. correctness with respect only to the pre- and post-execution states (input-output relation). However, Action Systems theory can also be extended to support stepwise refinement of reactive systems [7, 11, 15], in which case intermediate states are also considered.

2.3.2 Event-B

Another important formalism based on guarded commands is Event-B [1, 39]. It shares much of its theoretical background with the Action Systems formalism, as well as with the (classical) B-method [2]. It can also be seen as a successor of the B Action Systems framework [72]. Despite the many similarities, there are also a number of notable differences. For example, Event-B is based on sets (subsets of the state space) rather than predicates, and, therefore, *set transformers* are used instead of predicate transformers. However, this is mostly an illusionary difference, since the two approaches are effectively interchangeable. Another difference is the fact that Event-B, on purpose, has no fixed behavioural semantics, i.e. it does not take a stand on how the guarded commands are triggered. Furthermore, refinement is defined as a number of *proof obligations* [42] instead of the way it was defined in Section 2.2. This makes Event-B more flexible with respect to different use cases, since any behavioural semantics that is compatible with the proof obligations can be used. Typically, however, the same semantics as in action systems is used. It is also worth noting that Event-B does not support sequential composition, a fact that has consequences for the construction of guarded commands. Furthermore, the mathematical language of Event-B [58] is strictly defined, which is not the case for the Action Systems formalism. While this makes action systems more flexible, it also renders tool support difficult. For Event-B, tool support is currently provided by the open source Rodin tool [4, 3, 39]. It is based on the Eclipse platform [38], and additional functionality can be added through the means of plug-ins.

2.3.3 Terminology

We also want to point out some discrepancy in terminology between different guarded command based languages. In the Action Systems formalism, guarded commands are known as *actions*, whereas they are called *events* in Event-B. Furthermore, Event-B used the word "action" for the statement following the guard, which may cause confusion because the same word is used differently in Action Systems theory. In the rest of Part I of the thesis, we will typically use the denotation "guarded command" regardless of the formalism in question, and we will also otherwise try not to use ambiguous terms in this respect.

2.4 Scheduling

2.4.1 Non-deterministic Choice

Models expressed in the Action Systems formalism, or as loops in the Guarded Command Language, are scheduled by demonic non-deterministic choice between guarded commands. Such a behavioural semantics is also commonly used in Event-B, which is the case for the Event-B models studied in this thesis as well. The precise mathematical-logical semantics is given by the predicate transformers. However, an intuitive interpretation is that the developer has no control of which one of the guarded commands will be chosen for execution. To elaborate what this means, consider the following weakest precondition involving three guarded commands:

$$(g_1 \rightarrow S_1 \mathbin{[\!]} g_2 \rightarrow S_2 \mathbin{[\!]} g_3 \rightarrow S_3)(q) \tag{2.10}$$

According to the definition of demonic non-deterministic choice, this equals to $(g_1 \rightarrow S_1)(q) \wedge (g_2 \rightarrow S_2)(q) \wedge (g_3 \rightarrow S_3)(q)$. Consequently, in order for the composition to guarantee postcondition q, not only one or two, but all three guarded commands have to individually guarantee q. This can be seen as an adversary, or a demon, performing the choice, hence the name. In fact, there is also an *angelic non-deterministic choice*, according to which a composition achieves a given postcondition given that any one of the constituent guarded commands (or, in general, statements) does so. However, it is of less practical interest, and it is not *conjunctive*, which is a property required by many rules for algebraic reasoning about statements.

2.4.2 Parallel and Distributed Models

Since the guarded commands are typically located inside a **do od** loop, execution of the commands takes place repeatedly until all of them are disabled. Even though the choice is non-deterministic, once a guarded command has been chosen, it will be executed in its entirety. The procedure of choice and execution can be seen as sequentially repeated with only one command executing at a time. However, the semantics also allows for a parallel interpretation [7]. If several guarded commands are enabled and have no variables in common, they can be executed simultaneously. This convenient means of achieving parallelism is an argument for using guarded command based languages for the development of concurrent software. Furthermore, both the Action Systems formalism and Event-B support decomposition [5, 23, 10], which is useful when modelling distributed systems. Such decomposition can be of two basic types: either the components communicate via *shared variables*, or, if they have separate state spaces, they rely on *shared guarded commands*.

2.5 Problem Statement

2.5.1 Background

As we have seen, guarded command based languages have several advantages. First, they have a formal semantics supporting correctness preserving stepwise refinement. They also have a parallel interpretation that is suitable for modelling concurrent software. Furthermore, there are techniques for decomposition into sub-models. For Event-B, in particular, there is also tool support. Features such as these make languages based on guarded commands interesting for the development of parallel software in scenarios where correctness is of particular importance. These arguments constitute a compelling reason for studying and deploying guarded command based languages in the first place. However, these languages also carry a number of drawbacks that currently limit their usability. One such problem, in particular, is how to interpret demonic non-deterministic choice in terms of scheduling in real-world scenarios, in which case implementation on a physical computer is typically a goal. Analysing and providing answers to such questions in a way that is sound from a practical point of view, yet fully compatible with the underlying theoretical concepts, constitutes the core of this thesis.

2.5.2 Interpretation of Theoretical Concepts

The interpretation of demonic non-deterministic choice in practical terms of scheduling is not straightforward. Demonic non-deterministic choice is a rather theoretical concept, and it differs considerably from scheduling in traditional programming paradigms such as the imperative one. Apart from choice, there are also other theoretical modelling constructs that are typically not found in programming languages. A notable example is miraculous behaviour, which is not suitable for execution on a physical computer. However, the way it is typically used in guarded command based languages is as guards inside a **do od** loop, and as an assumption sequentially following the strong iteration in the definition of the loop. When used precisely in this way, the miraculous statements will never be executed, but they rather serve as a mechanism of restricting the demonic non-deterministic choice, as well as controlling loop termination. This ensures that only enabled commands can be chosen for execution, and that the loop does not terminate as long as at least one guarded command is enabled. Hence, the impact of miraculous constructs is reduced to scheduling considerations. Consequently, it is a part of the question about how demonic non-deterministic choice can be interpreted and implemented in terms of execution on a physical computer.

2.5.3 Questions of Particular Interest

An important question is whether interpretation and implementation of demonic non-deterministic choice is scenario specific, rendering guarded command based languages more suitable for certain types of scenarios than for others. Furthermore, a scheduling approach is not necessarily equally fit for all scenarios, leading to the question what scheduling techniques to be used in what cases. An interesting and fully relevant issue is also whether the guarded command based nature is to be preserved at the implementation stage, or whether models should be transformed to resemble programs written in traditional programming languages. Performance of the final software may also depend on the technique that has been used. We find that such issues are important for the adoption of guarded command based languages, and the rest of the thesis is dedicated to studying them, particularly in the context of parallel/ubiquitous scenarios.

2.6 Research Methods

The research methods used in the thesis largely boil down to the nature of computer science, and, in particular, to the sub-discipline we focus on. Predicates, weakest precondition predicate transformers as well as the refinement concept are deeply rooted in mathematics and logic. This calls for a mathematical and algebraic approach, in contrast to the empirical one that is characteristic of natural sciences. Some of the scheduling methods were, indeed, evaluated from a mathematical-logical correctness perspective.

However, despite its mathematical roots, there is also a practical aspect of formal methods and scheduling. The reason why formal methods are used in the first place is the development of computer programs, the performance of which may be difficult to evaluate from a purely theoretical point of view. Since factors such as communication overhead may also have a large impact, complexity analysis as used in algorithmics is also not applicable. Because of this, we found it motivated to perform an empirical study to evaluate the performance of a specific scheduling approach.

In addition to mathematical correctness and real-world performance, there is also an ease-of-use aspect. A scheduling method may be theoretically correct and perform well, yet be difficult to use from a developer's point of view. This property is difficult to quantify and evaluate from a strictly objective perspective. Nevertheless, we consider it an important aspect that we have paid attention to and which we discuss when relevant. We also shed more light on it when we compare the different scheduling techniques to each other.

Chapter 3

Contributions of the Original Publications

We will now focus on the research conducted in the original publications re-published in Part II. In addition to just describing the research of each paper in isolation (which can also be read in their respective abstracts), we here put emphasis on relating them to each other. We also shortly discuss to what degree the thesis author has been involved in each of the papers.

3.1 Paper I

F. Degerlund and K. Sere. "A Framework for Incorporating Trust into Formal Systems Development". In: *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Proceedings*. Ed. by Z. Liu C. B. Jones and J. Woodcock. Vol. 4711. Lecture Notes in Computer Science. Springer, 2007, pp. 154–168.

In this paper, we study a means of modelling *trust* in a social networking setting. The publication is based on a previous extended abstract [33]. The scenario involves a number of *entities* (or *agents*), that may interact with each other by the means of *transactions*. Each transaction may have some specific pre-requisites that must hold. Models are expressed in the Action Systems formalism, and transactions are represented in the form of guarded commands, whereby pre-requisites can be expressed in the guard. In the paper, we used a car sales scenario as an example, whereby e.g. the ability to afford the price of a car constituted such a pre-requisite. Trust aspects can be seen as additional entity-specific pre-requisites that must hold for a transaction to take place, in addition to the basic functional ones. There are several alternative frameworks for modelling trust, e.g. as proposed by Degerlund [31] or the *subjective logic* by Jøsang et al. [51].

We used the latter approach in the paper, and trust values representing the degree of trust between different entities was expressed using variables and the trust criteria were expressed in a special *co-ordination language.* The non-trust part of the system, expressed as an action system, and the trust aspects modelled in the co-ordination language can easily be merged into a standard action system, whereby trust criteria become part of the guards of the guarded commands representing transactions. We can conclude that the use of a guarded command based modelling language fits this particular use scenario well, since transactions and trust thresholds inherently exhibit an event-based nature.

Work division. The principles of the paper were developed jointly by both co-authors. Most of the writing was performed by Degerlund.

3.2 Paper II

F. Degerlund and K. Sere. “Refinement of Parallel Algorithms”. In: *Process Algebra for Parallel and Distributed Processing.* Ed. by M. Alexander and W. Gardner. Computational Science Series. Chapman & Hall / CRC Press (Taylor & Francis Group), 2008, pp. 77–96.

We now move from inherently event-based scenarios and focus on the modelling of parallel compute-intensive terminating programs. The approach we use in this paper has its roots in the PhD thesis of Sere [68]. We start from a sequential model and use algebraic rewrites to split it into separate parts that can be executed concurrently, and the methodology is demonstrated by applying it on a model for integer factorisation. The language used adheres to the Action Systems formalism, with the extension that we allow for multiple **do od** loops that can be sequentially composed. This extension is valid, since action systems can be seen as a canonical form of general statements. Refinement calculus is used to show that each rewriting step is a valid refinement step of the previous one. Note that we focus on correctness with respect to the input-output relation, i.e. total correctness. This guarantees that we achieve the desired end result, but at the same time allows us to flexibly restructure the inner workings of the model to be compatible with parallel execution. Rewriting is performed in such a way that we obtain sets of guarded commands, where the commands of different set should have as few variables in common as possible in order to maximise parallelism. The sets of guarded commands are intended to be executed on different computational nodes, interconnected according to a star topology, and scheduling takes place using the default non-deterministic choice. Note that while we keep in mind an intended mapping to computational nodes, we never state it explicitly. This is because we only operate on an abstract modelling level and we have no means of expressing implementation level details. Mappings contrary to the intended one do not lead to incorrect results, but performance will typically suffer since variable conflicts would impair the degree of parallelism.

Work division. The research was conducted in close co-operation by both authors. Degerlund produced large portions of the text, including most of the model listings.

3.3 Paper III

F. Degerlund, K. Sere and M. Waldén. "Implementation Issues Concerning the Action Systems Formalism". In: *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'07)*. Ed. by D. S. Munro, H. Shen, Q. Z. Sheng, H. Detmold, K. E. Falkner, C. Izu, P. D. Coddington, B. Alexander and S.-Q. Zheng. IEEE Computer Society Press, 2007, pp. 471–479.

In Paper II, we considered modelling parallel software for execution on computational nodes in a star topology. However, the solution was on quite a theoretical level. While the restructuring refinement steps led to a model designed to be suitable for parallel execution, more detailed implementation aspects were not considered. This paper, which has its roots in the ideas of an earlier extended abstract [34], focuses on the execution framework and how scheduling is implemented in practice. It does not concern the refinement process itself, which can be performed similarly to the method in Paper II or to the principles in Sere's thesis [68]. What is of particular importance is that the models have been developed in such a way that the guarded commands contain as few shared variables as possible. The basic principle in this paper is to use a central scheduler that implements the role of the theoretical non-deterministic demon. In addition to simply deciding what guarded commands to schedule at what time, it also delegates them to an appropriate computational node for execution. Scheduling decisions are taken based on factors such as variable conflicts between guarded commands, model designer decisions and fairness. Variable conflicts are expressed in a matrix, which the scheduler respects when deciding which events to schedule concurrently. The models designed may also provide a matrix containing information on what node(s) specific guarded command may be scheduled. Furthermore, the scheduler deploys a scheduling algorithm guaranteeing that all guarded commands will have a chance to be scheduled (provided that they are enabled), even though Action Systems theory as such does not deal with fairness. The scheduler is also responsible for upholding a record of the model state, and when scheduling a guarded command, it communicates the current values of the relevant variables to the executing node. A proof-of-concept version of the scheduler was developed in C++ [71] for use with code generated from the Atelier B [28] tool when used to model B Action Systems [72]. The MPI [56] framework allowed communication between the central scheduler and the computational nodes.

Work division. The scheduling principles were suggested primarily by Degerlund, but all three authors were involved in the work. The writing process was undertaken mostly by Degerlund.

3.4 Paper IV

F. Degerlund. *Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B*. Tech. rep. 1051. Turku Centre for Computer Science (TUCS), 2012.

Paper III relied on modelling in the Action Systems formalism and code generation was performed by the Atelier B tool. However, since Atelier B only supports classical B, the solution relied on using B Action Systems, which is an approach of expressing action systems in terms of classical B. To enable the use of a single formalism, we turned to Event-B and the Rodin tool. As part of Grönblom's master's thesis [41], a code generator was developed as a plug-in for the Rodin platform, and the scheduler of Paper III was updated to be compatible with the resulting code. The approach was also described in a technical report [32]. However, while this constituted a concrete framework for deployment, optimisation had not yet been considered, nor had the performance of the approach been evaluated in practice. Because of this, we introduced an optimisation approach and performed an empirical efficiency study in a conference article [30], of which Paper IV is an extended version. In the unoptimised version, each scheduled guarded command, is only executed once by a computational node before returning the updated variables to the central scheduler. This implies a considerable communication overhead. To counteract the problem, a repetition mechanism was introduced, allowing for the guarded command to be executed repeatedly a specific maximum number of times, or, until it disables itself. The scheduler implementation, which also contains code controlling the computational nodes, was adapted to support this mechanism, and a model for parallelised integer factorisation was used for benchmarking. The tests were performed on a multicore/multiprocessor machine, and we studied both the practical applicability of the method itself and the scalability when additional cores were used. The results were compared to a sequential C++ program that corresponded as closely as possible to the parallel model as far as the factorisation algorithm is concerned. While the sequential version obviously contained less overhead, it did not have the potential of speed-up from parallelism. Because of this, the sequential version was faster when the work load was low and the number of cores in use was low. However, the parallel version was considerably faster for larger workloads and a higher number of cores. We also found indications suggesting that the scheduling framework scales relatively well.

Work division. Degerlund is the sole author of this paper and the conference paper [30] on which it is based.

3.5 Paper V

P. Boström, F. Degerlund, K. Sere and M. Waldén. "Derivation of concurrent programs by stepwise scheduling of Event-B models". In: *Formal Aspects of Computing* (2012). DOI: 10.1007/s00165-012-0260-5.

The scheduling framework involving a central scheduler bears similarities to the non-deterministic choice of the modelling languages, and the developer does not explicitly have to take a stand on scheduling. However, such a scheduler needs computational resources at run-time, and sometimes the developer may desire to have more control over scheduling details. To cater for such situations, we proposed a decentralised approach involving explicit schedules in a workshop paper [21], of which Paper IV is an extended version. The idea is that instead of a dedicated scheduler taking decisions at run-time, the developer expresses an explicit schedule for each task/process. We rely on shared-variable model decomposition as in related work [5, 44] and use Event-B for modelling, but the schedules are given in a separate scheduling language containing e.g. sequential composition of guarded commands. What constitutes a particular challenge is that such scheduling deviates from the original non-deterministic choice of Event-B, yet the scheduled version must functionally adhere to the original specification. Because of this, schedules are introduced as refinement steps. We also note that miraculous behaviour may take place should a guarded command turn out to be disabled when the schedule mandates its execution. While this would strictly speaking constitute a valid refinement step, it is not desirable for practical reasons. Consequently, it also has to be shown that no new miraculous behaviour is introduced. Another challenge is that the scheduled system as a whole should be a refinement of the original model, i.e. it is not enough to consider each task in isolation, a problem we address by taking interference from other tasks into account. Consequently, there are several factors that have to be considered as part of schedule introduction, and to alleviate the burden on the developer, we provide pattern support for some common scheduling scenarios. To demonstrate our approach in practice, we schedule a model of the dining philosophers problem [47].

Work division. The idea was originated by Boström and Sere as an extension of Boström's previous work [20]. Further theoretical approaches were developed jointly by all authors. Most writing was performed by Degerlund and Boström, with some text contributed by Waldén and Sere. This division concerns the work resulting in both the underlying article [21] and this paper (Paper V).

Chapter 4

Analysis and Discussion

In this chapter, we focus on analysing the different scheduling approaches presented in the previous chapter and the original publications. We put particular emphasis on the respective advantages and drawbacks of each method, in what scenarios they are most useful, and to what degree they can be seen as mutually interchangeable for a given modelling scenario. Furthermore, we compare them to related work in order to highlight the contribution of the thesis and to position them in a wider research context. We also briefly discuss how well we have succeeded in addressing the problem statement of the thesis as given in Section 2.5, and what limitations remain.

The scheduling approaches we have considered can roughly be divided into three categories:

1. Event-driven scheduling
2. Run-time scheduling
3. Development-time scheduling

Using this classification, Paper I deals with approach number 1, papers II, III and IV use approach 2, whereas paper V relies on approach 3. We will now analyse each of the methods more closely.

4.1 Event-driven Scheduling

It is important to note that languages based on the Guarded Command Language have much in common with event-driven programming. This is because they contain a collection of instructions without explicit control flow, but rely on invocation from an external entity. It would be tempting to call them event-based languages, but that term is already reserved for process calculi, which clearly constitute a different class of languages. The key as to why both classes are related to "events" is that they represent two sides of the same coin. Whereas process calculi can be used for controlling invocations, guarded commands bear similarities to the event handlers in the event-based programming paradigm. The demonic non-deterministic choice typically used to invoke the guarded commands can be seen as a general placeholder for more elaborate control mechanisms. However, it is not the same as random choice, and it is important to be clear about this distinction. Whereas random choice would imply a specific distribution over time, demonic non-deterministic choice does not. Consequently, demonic non-deterministic choice is more general, and it is intuitively very close to the concept of spontaneous choices made by one or several agents (actors/entities) that act on their own behalf. This is similar to user interaction in e.g. graphical user interfaces, which are often designed in an event-based fashion. This constitutes a connection between guarded command based languages and event-based programming, and it also explains why ubiquitous social networking scenarios as discussed in Paper I can be conveniently modelled in a guarded command based language. The trust aspects that originally stem from the separate co-ordination language are easily merged with the guards of the guarded commands and fit well into the event-driven approach.

4.2 Run-time Scheduling

A key aspect of the scenarios discussed above is their event-driven nature, which facilitates both modelling in a guarded command based language as well as the scheduling. However, in other cases, there are no inherent agents that invoke the guarded commands and implement the demonic non-deterministic choice. For example, in compute-intensive scenarios such as those described in Papers II, III and IV, there is no explicit agent that interacts with the system. It may also not be obvious why such scenarios are modelled in a guarded command based language in the first place. However, in addition to their formal semantics and refinement support, the parallel interpretation of these languages constitutes a particular advantage when modelling systems intended for concurrent execution. The three papers dealing with the approach also specifically focus on parallel systems. Furthermore, they explicitly focus on terminating programs, which is in contrast to the scheduling scenario of Paper I. Again, this is a consequence of the nature of the scenarios modelled (compute-intensive result oriented vs ubiquitous user-centric).

4.2.1 Scheduling Framework

A crucial question is how to implement the non-deterministic demonic choice when there are no external agents available. The solution taken here is to use

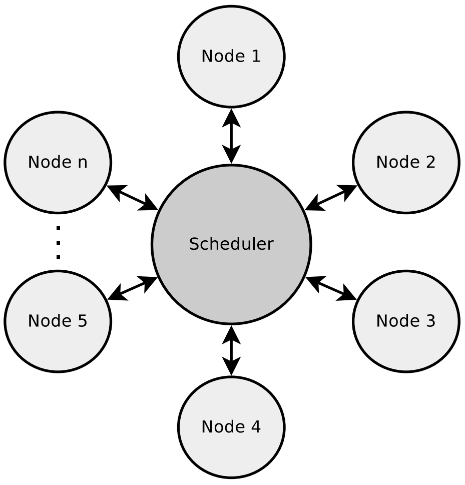


Figure 4.1: Central scheduler and slave tasks in the run-time scheduling scenario.

a dedicated piece of code, running as a task (process) of its own and scheduling guarded commands at run-time. When a guarded command is chosen, it will be executed in one of several slave tasks. The scenario is illustrated in Figure 4.1. Ideally, the physical platform should contain one computational unit, such as a processor or a processor core, for each slave process as well as one for the scheduler itself. The scheduler then takes advantage of the parallel interpretation of guarded command based languages, so that it may schedule independent guarded commands concurrently. Papers III and IV also specify low-level aspects such as the use of MPI as the intertask communication framework, how the variables are passed between the scheduling task and the slave ones, as well as how guarded commands are to be translated to programming language functions/methods. Taking a stand on such low-level details was important especially in Paper IV in which the scheduling efficiency of a model for integer factorisation was benchmarked. However, in general, other communication frameworks could also be used. Furthermore, it would be possible to implement a more light-weight version of the scheduling approach where a shared memory platform is assumed, whereby passing on variable values would be rendered unnecessary. However, this would obviously eliminate the possibility of executing the model on a cluster.

4.2.2 Related Techniques

Note that the run-time scheduling approach bears some similarities to the concept of *animation* as in the ProB [52], AnimB [57] and BRAMA [69] plug-ins for the Rodin platform. However, while both run-time scheduling and animation provide for a means of “running” the model by execution of enabled events, animation is not necessarily automatic but can also be manually controlled. Perhaps more importantly, the purpose of the run-time scheduling approach is very different from that of animation. Animation is typically performed in the development platform in order to test how the model behaves in practice. It can therefore be seen as a complement to consistency and refinement proofs as well as model checking. Run-time scheduling, on the other hand, is intended as a means of deploying the end product, i.e. it is not a part of the development process. Furthermore, animation typically does not support parallel execution.

It is also worth comparing our approach to the one used by Méry and Singh [55]. They have developed a tool, EB2ALL, for translating Event-B models to the programming languages C, C++, C# and Java. Scheduling of the resulting code takes place essentially by using a loop construct to invoke the (translated) guarded commands of the model. However, their primary focus seems to lie on the code generation part, and, to our knowledge, they do not provide support for concurrent execution. However, their code generator, in itself, is of interest. It can be seen as a more advanced version of an early Event-B code generator by Wright [74], which supports the most basic constructs. Wright’s work inspired the translator [32, 41] that was (partially) used in Paper IV, and EB2ALL could possibly constitute an even more advanced means of generating code for scheduling using our run-time scheduling approach.

4.3 Development-time Scheduling

In the above run-time scheduling approach, the demonic non-deterministic choice is implemented by introducing a dedicated scheduler, the role of which is to invoke the guarded commands. However, the introduction of a scheduling agent may seem like an artificial attempt to enforce an event-based approach for a scenario that is not inherently event-driven.

4.3.1 Towards Task-specific Schedules

An alternative way forward would be to focus on moving away from the event-focused nature of the guarded command based languages. Such an approach would eliminate the need for a separate scheduler, and the code would instead contain more elaborate control structures such as those typically associated with imperative programming languages. An important aspect of such a transition is to preserve the properties of the original specification. This can be achieved by introducing the constructs as one or several refinement steps. However, it may be inconvenient to introduce the control constructs on an ad-hoc basis directly on the set transformer level.

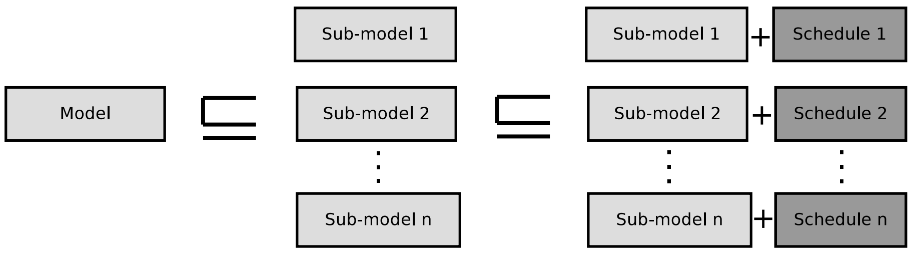


Figure 4.2: Development-time scheduling. The model is first decomposed into sub-models. Schedules are then introduced on the individual sub-models.

This is where schedules expressed in a separate scheduling language enter the picture. Furthermore, to constitute a viable alternative to the previously discussed run-time scheduling approach, parallel execution must be supported. This can be achieved by the developer giving a separate schedule for each task in the system. Prior to schedule introduction, the model has to be decomposed into sub-models, one for each task to be scheduled. Shared-variable decomposition can be performed both in the Action Systems formalism [10] and in Event-B [5]. Here, we have focused on the latter. A conceptual illustration of the development-time scheduling approach is given in Figure 4.2, showing a special case in which schedules are introduced in a single refinement step.

4.3.2 Comparison to Other Work

In the approach of Paper V, the schedules are given in a special purpose scheduling language, and this language is given a semantics through the means of a function mapping it to the corresponding set transformers. It then needs to be proven that the resulting set transformers constitute a refinement of the unscheduled model. As previously mentioned in the paper overview, patterns can be used to assist the developer. They can be seen as general proofs of commonly occurring scheduling scenarios, and the paper provides a few examples of such patterns. The approach of Paper V is an extension of an approach by Boström [20], in which a similar technique was used to schedule Event-B models sequentially. It also bears similarities to the scheduling frameworks of Hallerstede [43] and Iliasov [48]. The major improvement compared to Boström's sequential framework is the support for multiple tasks. Since the tasks communicate with each other using shared variables, they cannot be refined completely in isolation. This restriction also applies to scheduling, since schedule introduction involves refinement. The key here is to strike a balance by taking interference from other tasks into account in the form of *external variables* and *external events* (external guarded commands) [5], yet to allow for relatively independent refinement. The refinement approach of individual tasks of a parallel program is similar to what has been proposed by Hoang and Abrial [44]. However, explicit schedules constitute a complicating factor as compared to their work, where they use the standard demonic non-deterministic scheduling. A more comprehensive comparison to related work is given in Section 7 of Paper V.

4.4 Comparative Discussion

From the above discussion, we can see that the three scheduling approaches are quite different from each other, even though they all have the same goal, i.e. to schedule software developed using a guarded command based language. To emphasise their differences, we find it motivated to classify them as *scheduling paradigms*. As for terminology, it is also worth noting that the names "event-driven scheduling", "run-time scheduling" and "development-time scheduling" must be understood as the specific scheduling paradigms that we have analysed here. For example, "run-time scheduling" can generally be applied to any scheduling that is performed on-the-fly, whether it is sequential or parallel, and whether there is a separate scheduler or not, whereas "run-time scheduling" in this thesis has a very specific meaning.

4.4.1 Applicability

It is evident that the different scheduling paradigms are not equally well applicable in all possible cases. In fact, their respective suitability depends on the nature of the scenario. This gap is evident especially when comparing the event-driven scheduling to the two other paradigms, where practical applicability boils down to whether the scenario is inherently event-driven or not.

The choice between run-time and development-time scheduling is, in contrast, less obvious. Both paradigms constitute possible candidates for parallel models without inherent invoking agents. An important difference is that for run-time scheduling, a separate task is needed for the scheduler, in addition to those used by the model itself. For performance reasons, an extra hardware computational unit should preferably be reserved for this task. Furthermore, since scheduling is performed automatically and on-the-fly, the developer is not able to explicitly optimise scheduling, apart from simply limiting what guarded commands can be executed in what tasks. However, the developer can (and should) implicitly improve parallel schedulability by making sure that events have as few variables as possible in common.

In the development-time scheduling approach, no dedicated scheduling task is needed, and the developer has the possibility to write optimised schedules for each task. At first sight, it may seem as if there are only advantages as compared to run-time scheduling. However, designing appropriate schedules increases the burden on the developer, who has to foresee what events will be enabled at what point during execution. The schedules must also be proven correct, requiring additional effort from the developer. These problems do not exist in the run-time scheduling paradigm, since it is easy to check enabledness of the guarded commands at run-time. Another limitation of the development-time scheduling is that a shared memory platform is assumed, which is not the case for the run-time paradigm, if it is implemented, as proposed, using MPI. On the other hand, if the intended target platform indeed has shared memory, the run-time scheduling paradigm could be adapted to use e.g. POSIX threads instead of MPI, which could give a performance speed-up, at least in theory.

There is obviously not a clear-cut answer to which of these two paradigms is "better", but the above considerations should rather be evaluated in each individual case. An interesting prospect would be to use both paradigms for the same project. For example, run-time scheduling could be used for prototype development, whereas time would be invested in developing and proving explicit schedules for the final product.

4.4.2 Chronological Comparison

We also note that the three scheduling paradigms can be ordered chronologically with respect to how much they adhere to an event-driven approach. An illustration is given in Figure 4.3. The event-driven paradigm, obviously, presents the most event-driven characteristics of the three. Next, we have run-time scheduling, which is event-driven with an artificial invoking agent. The least event-driven paradigm is development-time scheduling, where explicit schedules replace the scheduling agent. However, while providing a means of classification, the chronological order does not necessarily correlate with the applicability of the paradigms.

4.5 Success, Completeness and Limitations

In Section 2.5, we gave a problem statement for the thesis and discussed questions to be answered. The core challenge was how to handle the theoretical nature of demonic non-deterministic choice for practical scheduling purposes, which is a question of particular interest in parallel scenarios. Our approach to tackling the problem has been to identify three distinct scheduling paradigms and to explore their respective merits and applicability. These methods have been thoroughly discussed and we have demonstrated how they can be applied.

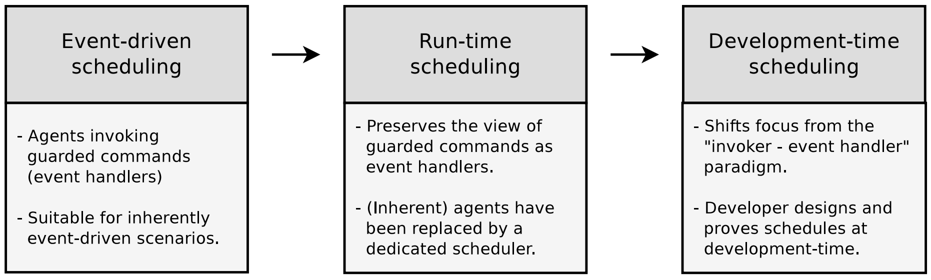

Event-driven scheduling
- Agents invoking guarded commands (event handlers)
- Suitable for inherently event-driven scenarios.
Run-time scheduling
- Preserves the view of guarded commands as event handlers.
- (Inherent) agents have been replaced by a dedicated scheduler.
Development-time scheduling
- Shifts focus from the "invoker - event handler" paradigm.
- Developer designs and proves schedules at development-time.

Figure 4.3: Chronological comparison indicating relative differences between the scheduling paradigms.

We have also compared the scheduling paradigms to each other and briefly discussed which one to choose for what kind of scenarios. However, an interesting question is how well the paradigms, when taken as a whole, cover the whole array of conceivable scheduling needs. A complicating factor is that the scheduling paradigms have been chosen on the basis of practical considerations rather than strict mathematical-logical reasoning, making it more difficult to draw conclusions on their “completeness” on the whole. Despite this, some interesting facts can be noted. For example, from a strictly computational point of view, any model that can be translated can also be executed according to the run-time scheduling paradigm. Constructs that are not currently supported by code generation cannot be used, though, and the resulting code may be very inefficient if the method is applied on models that are less than ideal for automatic scheduling. In such cases, re-modelling (e.g. optimising for parallelism and repeated execution of events) or switching to development-time scheduling may be appropriate. Furthermore, applying the computationally oriented run-time scheduling paradigm to event-driven scenarios does not make much sense in practice, since the guarded commands would not be associated with external events, missing what constitutes the purpose of such models in the first place. Despite the fact that many, or most, models can be scheduled using one of the three paradigms, there may be models for which none of them is ideal. A problematic scenario would be a compute-intensive model that is difficult to optimise or parallelise for automatic run-time scheduling, and for which the proof obligations involved in development-time scheduling are hard to prove. However, sometimes such difficulties may also be due to inappropriate modelling or algorithms rather than limitations of the scheduling paradigms.

On the whole, we believe that while challenges and limitations remain, our research has been successful in providing additional insight into the problem of implementing and scheduling models based on guarded commands. Our work indicates that the proposed solutions may be viable and have the potential as a foundation for further research and practical deployment.

Chapter 5

Conclusions

In this thesis, we have studied practical scheduling aspects of languages with their roots in Dijkstra's Guarded Command Language. While we have focused primarily on the Action Systems formalism and Event-B, we believe that both the challenges and solutions could be extended to other languages based on guarded commands. This class of languages provides a powerful means of reasoning about models, largely thanks to the weakest precondition predicate transformer semantics and the Refinement Calculus. The behavioural semantics of models in these languages relies on demonic non-deterministic choice between enabled guarded commands, i.e. those guarded commands whose guards evaluated to true. This theoretical scheduling principle also has an impact on how models are to be expressed, and it may or may not be convenient for representing a given scenario. An equally interesting aspect is how the non-deterministic scheduling works out when an abstract model is implemented and executed on physical computers. Since guarded command based languages are particularly suitable for concurrent scenarios, we have focused on scheduling of parallel and ubiquitous systems.

We first studied a ubiquitous scenario in which scheduling took place solely based on external agents triggering the guards. Additional scheduling criteria, reflecting social trust between agents, were modelled in a separate co-ordination language. These policies were then imposed onto the main model as part of the guards. This approach turned out to be successful, but it cannot be used in more general cases. The ubiquitous scenario we studied was inherently event-driven, in which case the agents conveniently play the role of the non-deterministic demon.

In the general case, there are no inherent agents that invoke the guarded commands. One approach to addressing this problem is to introduce a dedicated scheduler. We focused on parallel, compute-intensive scenarios and suggest a scheduling task, or process, that invokes guarded commands in computational slave tasks. It is the responsibility of the scheduler not to delegate conflicting guarded commands for concurrent execution. Furthermore, the scheduler keeps track of the global state space of the model, and variables are communicated between the scheduler and the slave nodes as needed. In order to avoid excessive communication overhead, we proposed an optimisation, according to which the slaves under certain circumstances may repeat guarded command execution without the involvement of the scheduler. To evaluate practical applicability of the approach, we performed an empirical performance study on a multicore/multiprocessor computer using the MPI message passing library for inter-task communication. We found that the approach performed and scaled well in our test case model. However, certain granularity parameters has to be appropriately set in order to achieve the good results.

Finally, we considered a scheduling paradigm in which schedules for the tasks are manually designed at development-time. This is in contrast to the automatic run-time scheduling discussed above. Here, we also assumed that the variables are stored in a shared memory. The schedules are expressed in a dedicated scheduling language, and they are introduced as refinement steps onto sub-models, which result from shared-variable decomposition of the initial model. To assist the developer, we proposed a pattern-based approach to showing that the scheduled task is, indeed, a refinement of the original sub-model. A complicating factor is interference from other tasks. This aspect is taken into account in the correctness proofs, and it manifests as external guarded commands when operating at a low level. However, efforts were made to abstract it away from the developer, and it is not visible on the scheduling language level.

In conclusion, we have considered three different paradigms for scheduling models expressed in guarded command based languages. We have compared these approaches to each other as well as to related methods in the literature. The methods are quite different from each other, and they have their respective advantages and drawbacks. Furthermore, they are not equally suitable for all cases, but what can be considered the most appropriate scheduling paradigm depends on the scenario in question. We believe that practical scheduling considerations are important for the wider adoption of guarded command based languages, and it is our hope that the contributions of this thesis will constitute one further step in that direction.

Bibliography

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

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

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta and L. Voisin. “Rodin: an open toolset for modelling and reasoning in Event-B”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.6 (2010), pp. 447–466.

[4] J.-R. Abrial, M. Butler, S. Hallerstede and L. Voisin. “An Open Extensible Tool Environment for Event-B”. In: *Formal Methods and Software Engineering.* Ed. by Z. Liu and J. He. Vol. 4260. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 588–605.

[5] J.-R. Abrial and S. Hallerstede. “Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B”. In: *Fundamenta Informaticae* 77.1-2 (2007), pp. 1–28.

[6] R. J. R. Back. “On the Correctness of Refinement Steps in Program Development”. Report A-1978-4. PhD thesis. University of Helsinki, Department of Computer Science, 1978.

[7] R. J. R. Back. “Refinement calculus, part II: Parallel and reactive programs”. In: *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness.* Ed. by J. de Bakker, W. de Roever and G. Rozenberg. Vol. 430. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1990, pp. 67–93.

[8] R. J. R. Back and R. Kurki-Suonio. “Decentralization of process nets with centralized control”. In: *Proceedings of the second annual ACM symposium on Principles of distributed computing.* ACM, 1983, pp. 131–142.

[9] R. J. R. Back and R. Kurki-Suonio. “Distributed Cooperation with Action Systems”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10.4 (Oct. 1988), pp. 513–554.

[10] R. J. R. Back and K. Sere. “From Action Systems to Modular Systems”. In: *Formal Methods Europe (FME’94)*. Lecture Notes in Computer Science. Springer-Verlag, 1994, pp. 1–25.

[11] R. J. R. Back and K. Sere. “Superposition Refinement of Reactive Systems”. In: *Formal Aspects of Computing* 8 (3 1996), pp. 324–346.

[12] R. J. R. Back and J. von Wright. “Reasoning algebraically about loops”. In: *Acta Informatica* 36 (1999), pp. 295–334.

[13] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. First edition. Springer, 1998.

[14] R. J. R. Back and J. von Wright. “Refinement calculus, part I: Sequential nondeterministic programs”. In: *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*. Ed. by J. de Bakker, W. de Roever and G. Rozenberg. Vol. 430. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1990, pp. 42–66.

[15] R. J. R. Back and J. von Wright. “Trace Refinement of Action Systems”. In: *CONCUR ’94: Concurrency Theory*. Ed. by B. Jonsson and J. Parrow. Vol. 836. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1994, pp. 367–384.

[16] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[17] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham et al. *Manifesto for Agile Software Development*. 2001. URL: `http://agilemanifesto.org/`.

[18] J. Bicarregui, J. Fitzgerald, P. Larsen and J. Woodcock. “Industrial Practice in Formal Methods: A Review”. In: *FM 2009: Formal Methods*. Ed. by A. Cavalcanti and D. Dams. Vol. 5850. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 810–813.

[19] D. Bjørner and C. B. Jones, eds. *The Vienna Development Method: The Meta-Language*. Vol. 61. Lecture Notes in Computer Science. Springer, 1978.

[20] P. Boström. “Creating Sequential Programs from Event-B Models”. In: *Integrated Formal Methods: 8th International Conference (IFM 2010)*. Ed. by D. Méry and S. Merz. Vol. 6396. Lecture Notes in Computer Science. Springer, 2010.

[21] P. Boström, F. Degerlund, K. Sere and M. Waldén. “Concurrent Scheduling of Event-B Models”. In: *Proceedings 15th International Refinement Workshop*. Ed. by J. Derrick, E. A. Boiten and S. Reeves. Vol. 55. Electronic Proceedings in Theoretical Computer Science (EPTCS). Open Publishing Association, 2011, pp. 166–182.

[22] S. M. Brien and J. E. Nicholls. *Z Base Standard Version 1.0.* Technical Monograph PRG-107. 1992.

[23] M. Butler. “Decomposition Structures for Event-B”. In: *Integrated Formal Methods.* Ed. by M. Leuschel and H. Wehrheim. Vol. 5423. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 20–38.

[24] J. N. Buxton and B. Randell, eds. *Software Engineering Techniques.* Report on the NATO Software Engineering Conference 1969. NATO Science Committee, 1970.

[25] A. Cavalcanti, A. Sampaio and J. Woodcock. “A Refinement Strategy for *Circus*”. In: *Formal Aspects of Computing* 15 (2 2003), pp. 146–181.

[26] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[27] E. Clarke and E. Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs Workshop 1981.* Vol. 131. Lecture Notes in Computer Science. Springer, 1982, pp. 52–71.

[28] ClearSy System Engineering. *Atelier B web site.* URL: `http://www.atelierb.eu/`.

[29] *Creol web site.* URL: `http://heim.ifi.uio.no/~creol/`.

[30] F. Degerlund. “Scheduling of Compute-Intensive Code Generated from Event-B Models: An Empirical Efficiency Study”. In: *Proceedings of Distributed Applications and Interoperable Systems (DAIS) 2012.* Ed. by K. Göschka and S. Haridi. Vol. 7272. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 177–184.

[31] F. Degerlund. “Trust Mass, Volume and Density - a Novel Approach to Reasoning about Trust”. In: *Proceedings of the 2nd International Workshop on Security and Trust Management (STM 2006).* Ed. by S. Etalle and P. Samarati. Vol. 179. Electronic Notes in Theoretical Computer Science. Elsevier, 2007, pp. 87–96.

[32] F. Degerlund, R. Grönblom and K. Sere. *Code Generation and Scheduling of Event-B Models.* Tech. rep. 1027. Turku Centre for Computer Science (TUCS), 2011.

[33] F. Degerlund and K. Sere. “A Framework for Incorporating Trust into the Action Systems Formalism”. In: *Proceedings for the 18th Nordic Workshop on Programming Theory (NWPT’06), Reykjavík, Iceland, 18-20 October, 2006.* 2006.

[34] F. Degerlund, M. Waldén and K. Sere. “Implementation Issues Concerning the Action Systems Formalism”. In: *17th Nordic Workshop on Programming Theory, Presentation Abstracts.* DIKU, University of Copenhagen, 2005, pp. 59–61.

[35] E. W. Dijkstra. “A constructive approach to the problem of program correctness”. In: *BIT Numerical Mathematics* 8 (3 1968), pp. 174–186.

[36] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[37] E. W. Dijkstra. “Guarded commands, nondeterminacy and formal derivation of programs”. In: *Communications of the ACM* 18.8 (Aug. 1975), pp. 453–457.

[38] *Eclipse platform.* URL: `http://www.eclipse.org/`.

[39] *Event-B and Rodin platform web site.* URL: `http://www.event-b.org/`.

[40] R. W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science* 19 (1967), pp. 19–32.

[41] R. Grönblom. “A Framework for Code Generation and Parallel Execution of Event-B Models”. Master’s thesis. Åbo Akademi University, 2009.

[42] S. Hallerstede. “On the Purpose of Event-B Proof Obligations”. In: *Abstract State Machines, B and Z.* Ed. by E. Börger, M. Butler, J. Bowen and P. Boca. Vol. 5238. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 125–138.

[43] S. Hallerstede. “Structured Event-B models and proofs”. In: *Abstract State Machines, B and Z.* Vol. 5977. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 273–286.

[44] T. S. Hoang and J.-R. Abrial. “Event-B decomposition for parallel programs”. In: *ABZ 2010.* Vol. 5977. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 319–333.

[45] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (Oct. 1969).

[46] C. A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21 (8 Aug. 1978), pp. 666–677.

[47] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[48] A. Iliasov. *On Event-B and control flow.* Tech. rep. CS-TR-1159. School of Computing Science, Newcastle University, 2009.

[49] E. B. Johnsen and O. Owe. “An Asynchronous Communication Model for Distributed Concurrent Objects”. In: *Software and Systems Modeling* 6 (1 2007), pp. 39–58.

[50] E. B. Johnsen, O. Owe and M. Arnestad. "Combining Active and Reactive Behavior in Concurrent Objects". In: *Proceedings of the Norwegian Informatics Conference (NIK'03)*. Ed. by D. Langmyhr. Tapir Academic Publisher, Nov. 2003, pp. 193–204.

[51] A. Jøsang and S. J. Knapskog. "A metric for trusted systems". In: *Proceedings of the 21st National Security Conference*. NSA, 1998.

[52] M. Leuschel and M. Butler. "ProB: an automated analysis toolset for the B method". In: *International Journal of Software Tools for Technology Transfer (STTT)* 10 (2 Feb. 2008), pp. 185–203.

[53] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (July 1993), pp. 18–41.

[54] J. L. Lions. *ARIANE 5 Flight 501 Failure*. Report by the Inquiry Board. 1996. URL: `http://www.di.unito.it/~damiani/ariane5rep.html`.

[55] D. Méry and N. K. Singh. "Automatic Code Generation from Event-B Models". In: *Proceedings of the Second Symposium on Information and Communication Technology*. SoICT '11. ACM, 2011, pp. 179–188.

[56] *Message Passing Interface Forum*. URL: `http://www.mpi-forum.org/`.

[57] C. Métayer. *AnimB web site*. URL: `http://www.animb.org/`.

[58] C. Métayer and L. Voisin. *The Event-B Mathematical Language*. 2009. URL: `http://deploy-eprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf`.

[59] B. Meyer. *Eiffel: The Language*. First edition. Prentice Hall, 1991.

[60] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[61] R. Milner. *Communicating and mobile systems: the pi-calculus*. First edition. Cambridge University Press, 1999.

[62] C. Morgan. *Programming from Specification*. Second edition. Prentice Hall, 1994.

[63] J. M. Morris. "A theoretical basis for stepwise refinement and the programming calculus". In: *Science of Computer Programming* 9.3 (Dec. 1987), pp. 287–306.

[64] G. J. Myers. *The Art of Software Testing*. First edition. Wiley, 1979.

[65] D. Parnas. "Really Rethinking 'Formal Methods'". In: *Computer* 43 (2010), pp. 28–34.

[66] A. Pnueli. "The temporal logic of programs". In: *Proceedings of 18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57.

[67] W. Royce. “Managing the Development of Large Software Systems”. In: *Proceedings of IEEE WESCON*. IEEE, 1970.

[68] K. Sere. “Stepwise Derivation of Parallel Algorithms”. PhD thesis. Åbo Akademi University, 1990.

[69] T. Servat. “BRAMA: A New Graphic Animation Tool for B Models”. In: *B 2007: Formal Specification and Development in B*. Ed. by J. Julliand and O. Kouchnarenko. Vol. 4355. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 274–276.

[70] J. Spivey. *The Z Notation: A Referene Manual*. Prentice-Hall, 1992.

[71] B. Stroustrup. *The C++ Programming Language*. Third edition. Addison-Wesley Professional, 1997.

[72] M. Waldén and K. Sere. “Reasoning about Action Systems using the B-Method”. In: *Formal Methods in System Design* 13 (1 1998), pp. 5–35.

[73] N. Wirth. “Program development by stepwise refinement”. In: *Communications of the ACM* 14 (4 Apr. 1971), pp. 221–227.

[74] S. Wright. “Using EventB to Create a Virtual Machine Instruction Set Architecture”. In: *Abstract State Machines, B and Z* (2008), pp. 265–279.

Part II

Original Publications

Paper I

A Framework for Incorporating Trust into Formal Systems Development

Fredrik Degerlund and Kaisa Sere

Full original reference:

F. Degerlund and K. Sere. “A Framework for Incorporating Trust into Formal Systems Development”. In: *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Proceedings*. Ed. by Z. Liu C. B. Jones and J. Woodcock. Vol. 4711. Lecture Notes in Computer Science. Springer, 2007, pp. 154–168.

N.B. This paper is not included in the electronic edition of the thesis.

Paper II

Refinement of Parallel Algorithms

Fredrik Degerlund and Kaisa Sere

Full original reference:

F. Degerlund and K. Sere. "Refinement of Parallel Algorithms". In: *Process Algebra for Parallel and Distributed Processing.* Ed. by M. Alexander and W. Gardner. Computational Science Series. Chapman & Hall / CRC Press (Taylor & Francis Group), 2008, pp. 77–96.

Paper III

Implementation Issues Concerning the Action Systems Formalism

Fredrik Degerlund, Kaisa Sere and Marina Waldén

Full original reference:

F. Degerlund, K. Sere and M. Waldén. “Implementation Issues Concerning the Action Systems Formalism”. In: *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT’07)*. Ed. by D. S. Munro, H. Shen, Q. Z. Sheng, H. Detmold, K. E. Falkner, C. Izu, P. D. Coddington, B. Alexander and S.-Q. Zheng. IEEE Computer Society Press, 2007, pp. 471–479.

N.B. This paper is not included in the electronic edition of the thesis.

Paper IV

Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B

Fredrik Degerlund

Full original reference:

F. Degerlund. *Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B*. Tech. rep. 1051. Turku Centre for Computer Science (TUCS), 2012.

The paper is based on the following work:

F. Degerlund. "Scheduling of Compute-Intensive Code Generated from Event-B Models: An Empirical Efficiency Study". In: *Proceedings of Distributed Applications and Interoperable Systems (DAIS) 2012*. Ed. by K. Göschka and S. Haridi. Vol. 7272. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 177–184.

Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B

Fredrik Degerlund
Åbo Akademi University, Department of Information Technologies
Joukahainengatan 3-5, FIN-20520 Åbo/Turku, Finland
`fredrik.degerlund@abo.fi`

TUCS Technical Report
No 1051, August 2012

Abstract

Event-B is a tool-supported specification language that can be used e.g. for the modelling of concurrent programs. This calls for code generation and a means of executing the resulting code. One approach is to preserve the original event-based nature of the model and use a run-time scheduler and message passing to execute the translated events on different computational nodes. While constituting a straightforward method, it involves considerable communication overhead, a problem aggravated by the fine-grained nature of events in Event-B. In this paper, we consider the efficiency of such a solution when applied to a compute-intensive model. In order to mitigate overhead, we also use a method allowing computational nodes to repeat event execution without the involvement of the scheduler. To find out under what circumstances the approach performs most efficiently, we perform an empirical study with different parameters.

Keywords: Parallel computing, Event-B, Scheduling, Message passing, Efficiency

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Event-B [2] is a formal modelling language based on set transformers and the stepwise refinement approach. While designed for full-system modelling, it can also be used for correct-by-construction software development. Event-B also has a parallel interpretation, which allows for the modelling of concurrent systems. Tool support for Event-B has been achieved through the open-source Rodin platform [3, 4, 27], to which further functionality can be added in the form of plug-ins.

Code generation from Event-B can be achieved in a number of different ways. A straightforward approach that preserves the event nature has been proposed in [12, 18], for which a preliminary plug-in has been developed. In this approach, the model is translated into a C++ class, where events are directly translated into methods. The methods are invoked using a separate scheduler, which in turn deploys the MPI (Message Passing Interface) [24] library to achieve parallel execution on a multi-core/multi-processor system, or even on a cluster. This solution has the advantage that code execution very closely reflects the operating mechanisms of the Event-B model. An additional benefit is that it also does not require the developer to take a stand on specific schedules and prove that they are compatible with the original model.

However, this approach has a potentially serious drawback in the amount of overhead introduced by the scheduler and the MPI communication. Due to the practical nature of communication overhead, we recognise that it is difficult to evaluate the impact from a strictly mathematical-logical perspective. The purpose of this paper is, instead, to evaluate the viability of the scheduling approach by performing an empirical study. Since preliminary tests indicate that the overhead is unacceptably large, we propose a means of repeating execution of events without the involvement of the scheduler. The repetitive approach is implemented as part of the scheduling platform, and we let a factorisation model serve as a testbed for benchmarking. This technical report constitutes an extended version of a previously published conference paper [11]. We here provide additional background information as well as a more detailed description of our research than in the original article.

The rest of the paper is structured as follows. We first discuss related work in Section 2. In Section 3, we present background information on the Event-B formalism to the extent needed for understanding this paper. We also discuss how the models can be translated into a programming language (C++). Section 4 is dedicated to concurrent scheduling of models. We also deal with communication overhead and propose a repeating approach to improve efficiency. In Section 5, we present the factorisation model that serves as the testbed for our study, whereas we in Section 6 discuss how the actual benchmarking takes place. We give a number of test configurations that we have used for the test runs, after which we present the resulting execution times as well as an interpretation thereof. Finally, we sum up the paper and draw conclusions in Section 7.

2 Related Work

Unlike the classical B method [1], which focuses on a *correct-by-construction* approach, Event-B [2] was designed with system-level modelling in mind, but it can also be used for pure software development. The formalism has its roots in B Action Systems [30], based on the Action Systems formalism [6], which has been used e.g. for the derivation of parallel algorithms [28]. As a result, Event-B is also suitable for modelling of parallel software. The use of Event-B is facilitated by the Rodin platform [3, 4, 27], which provides tool support for the formalism. Rodin is based on the Eclipse framework [14], and custom plug-ins can also be used in the platform to provide additional functionality.

The Event-B scheduling approach we evaluate in the paper is based on [12, 18], which in turn has its roots in [13]. It is superficially related to the concept of animation as in the ProB [22], AnimB [25] and Brama [29] plug-ins for the Rodin platform. However, animation can be seen as a supplementary methodology during the modelling and development stage, while we (as in [12, 13, 18]) use automated scheduling as a means of executing the final code generated from the model. Furthermore, parallelism is typically not supported in animation, since the primary goal of animation is to analyse models instead of achieving efficient execution.

Another approach to scheduling of Event-B models has been taken in papers [21], [19] and [8]. The basic idea is to provide the models with explicit (sequential) control flow information expressed in dedicated scheduling languages. The developer then has to prove that the desired control flow is correct with respect to the typical Event-B behavioural semantics discussed in Section 3. This kind of scheduling can also be extended to handle parallelism [9]. However, an important difference as compared to the method we study is that scheduling decisions are taken and proven correct by the developer at the modelling stage. The approach we explore can instead be regarded as *on-the-fly* scheduling, where the scheduler takes scheduling decisions during *run-time* based on the current state. This eliminates the need for explicit schedule design and associated proofs, but may, on the other hand, induce a performance penalty.

A means of scheduling is also proposed in [23] for use with code obtained by the Event-B to C/C++/C#/Java code generator EB2ALL, which the authors present in the paper. However, to our knowledge, it supports only sequential execution, and therefore operates in a setting different from the one we consider here. An approach that does support parallelism is given in [15], where Event-B models are translated into Java for concurrent execution. The schedules are expressed by the model developer in a dedicated language called OCB (Object-oriented Concurrent B). In that sense, it bears similarities to the developer-scheduled approaches discussed above, in contrast to an on-the-fly approach. The method has also more recently been adapted [16] for use with the Ada language.

3 Event-B and Code Generation

3.1 The Event-B Formalism

Models in Event-B consist of *static* and *dynamic* parts, denoted *contexts* and *machines*, respectively. Contexts may contain e.g. *constants*, *carrier sets* and *axioms*, and can be used by one or several machines. Machines, in turn, contain elements such as *variables*, *events* and *invariants*. The variables v form the state space of the model, whereas events model atomic state updates. There is also a special *initialisation* event that gives initial values to the variables. The invariant $I(v)$ is used to assign types to the variables, as well as to restrict the valid state space. Consequently, the initialisation event must *establish* the invariant, whereas the rest of the events must *preserve* it.

Each event, except for the initialisation, contains a *guard* $G(v)$ and an *action* $v :| A(v, v')$. The guard contains a condition that must hold in order for the event to be allowed to take place, whereby the event is said to be *enabled*. The action describes how the state space is to be updated once the event is enabled and triggered. An event can be expressed in the following general form [20]:

$$E \triangleq \textbf{when } G(v) \textbf{ then } v :| A(v, v') \textbf{ end}$$

Here, v and v' represent the variables before and after the event has taken place, respectively. The operator :| represents non-deterministic assignment, whereby $v :| A(v, v')$ intuitively means that the variables v are updated in such a way that the *before-after predicate* $A(v, v')$ holds. A special case of the non-deterministic assignment operator is the deterministic assignment, :=, which closely resembles the assignment operator in standard programming languages. Note that the initialisation event is an exception, containing only an action but no guard. It also does not depend on a previous state.

Refinement [5, 7, 31] is a key concept in Event-B, enabling models to be developed in a stepwise manner. The idea is to achieve a chain of models, beginning from an abstract one and gradually turning it into more concrete ones. For each step, it must be shown that the new model is correct with respect to the previous one. We omit a detailed description of refinement in this paper, since we only focus on the last refinement step, which is the one to be converted to program code.

Event-B does not mandate any specific behavioural semantics. Instead, it defines a number of proof obligations, and any semantics compatible with them can be used. Typically, the same behavioural semantics as in the Action Systems formalism is deployed, and that one has also been used in this paper. First, the initialisation event is executed, after which the rest of the execution can be thought of as the events of the machine residing inside a loop. In each iteration, any enabled event is non-deterministically chosen for execution, and the loop only terminates when no event is enabled any longer. This can be interpreted as a deadlock situation in control systems, but for the input-output focused models we are interested in, it corresponds to termination.

3.2 Code Generation

Event-B does not specify how to generate executable code from models, and the Rodin tool in its basic form cannot translate models into a programming language without the use of extensions. However, a number of different approaches have been proposed. In [32], a code generator plug-in was developed. It was mainly intended for use as part of a virtual machine project, and supported translation of the most important Event-B constructs. This approach was taken a step further towards a more general-purpose tool, albeit an experimental one, in [12, 18]. The model first has to be refined according to the Event-B refinement rules (e.g. using the Rodin tool) until the events only contain concrete constructs that have direct equivalents in C++. The guard of the event is translated into a method returning a boolean value reflecting enabledness, whereas the action results in a separate method containing the C++ equivalent of its assignments. The idea was that the resulting methods could be invoked by an accompanying scheduler.

The testbed model (see Section 5) we benchmark in this paper (Section 6) is based upon a model originally used in [12, 18], and the translated code thereof. The model has, however, been amended in ways that could not be handled by the translation plug-in, and the code used for in this paper has, to a certain degree, been translated manually. Even though we here rely on manually generated code for evaluation of the scheduling approach, the process is time-consuming and error-prone. Due to the latter, in particular, manual translation may negate the correctness benefits of formal methods and does not constitute a realistic option for use in industrial projects. A possible path forward would be further development of the code generator of [12, 18]. An alternative approach would be to use the translation tool EB2ALL, even though adaptations would have to be made for the resulting code to be in a form compatible with the desired scheduling as discussed in the next section.

4 Scheduling

4.1 Scheduling Platform

When an Event-B model has been translated into C++ code, a means of scheduling the resulting code is required. Since we in this paper are interested in evaluating the viability of run-time scheduling, we need a scheduling platform that can invoke the methods that have been translated from the events. A prototype version of such a scheduler, called ELSA, was developed in [13] for running code generated from the Atelier B tool [10] when used for developing B Action Systems. The goal was to be able to execute the code of compute-intensive models in parallel on a multi-processor computer or a cluster using the MPI framework. In [12, 18], ELSA was adapted for use with code translated from Event-B models using a plug-in developed as part of the same research. We use this Event-B compatible version of ELSA for the evaluations performed in this paper, but we have improved it further in a number of ways, e.g. to handle 64-bit integers and to support repetition of events as presented in Section 4.2.

The scheduler code, which is written in C++, technically runs as part of both the scheduling process and a number of slave processes. The code takes a separate execution path on the scheduling process than on the event-executing slave processes, reflecting the different roles they play. The processes are mapped to physical processors or cores by the MPI framework, which the scheduling software uses for all inter-process communication. Communication takes place according to a star topology with scheduling process is in the centre, delegating event execution to the slaves. The scheduling process keeps track of the state space of the model, and when delegating an event for execution, it submits the current values of the variables involved to the slave. When the slave has executed the event, it returns the updated values of the variables to the scheduling process. To avoid conflicts, events that have variables in common must not be scheduled in parallel. It is also the responsibility of the scheduler to verify that events are enabled prior to delegating them. Enabledness is easy to check, since the guards are translated as boolean functions separate from the event actions. Though much simplified, the workings of the scheduling process can be explained as checking events for enabledness and delegating them for execution to slaves that are currently not processing any other events. This takes place until no events are enabled, whereby the scheduler terminates execution. A more detailed description of the scheduling algorithm can be found in [12, 18].

4.2 Repeated Execution of Events

The scheduler in its basic form, as described above, has a practical problem that needs to be tackled. After initial testing, it became evident that the overhead involved outweighs the benefits that parallelism can provide, resulting in poor execution times. The heart of the problem is not only the overhead in itself, but it becomes particularly problematic when combined with the fine-grained nature of Event-B events (or the corresponding C++ code). Events cannot contain structures such as sequential composition or loops, and complex behaviour instead has to be modelled in an alternative way, such as by repeated execution of events.

The scheduling approach above would imply that if an event is executed several times in a row, the scheduling processes would be involved in every invocation, resulting in excessive overhead. For this reason, we have amended the scheduling platform so that the slave processes may execute an event several times on their own. Before the scheduling process first delegates an event, it verifies the enabledness and passes on the values of the variables to the slave process as previously described. However, after execution, the slave checks whether the event is still enabled. If that is the case, it may run it again without any involvement of the scheduling process. This procedure may take place several times, until the event has been executed at most *REPEAT* times (including the initial execution delegated by the scheduling process), after which the updated variable values are reported to the central scheduler. The constant *REPEAT* can be seen as a parameter of the scheduling platform, and it applies to all slave processes and, in principle, to all events. However, since events may disable themselves even after only one or a few consecutive executions, *REPEAT* is to be seen as an upper limit. Also note that an event does not automatically become disabled after being executed *REPEAT* times, but to continue running it, it must once again be chosen for execution by the scheduling process. In fact, the repetition mechanism has no impact on the enabling/disabling of events, and it operates within the limits of the behavioural semantics as described in Section 3.1.

5 Testbed Model

A suitable testbed model for our study should be compute-intensive, easily parallelisable, convenient to express, and, for generality, as representative as possible of how other high performance computation models would be expressed in Event-B. The generality of the model is particularly important, since our goal is to draw as universal conclusions as possible on the viability of the scheduling approach. We find that an integer factorisation example given in [12, 18] for the most part fulfils these requirements. However, since we have made improvements to the scheduling approach as compared to [12, 18], especially by introducing repetition of events, we have also revised the model accordingly.

The goal of the model is to find a factor of a given integer n, such that it is greater than or equal to 2 and less than n. However, if n is a prime number, the result reported will be n itself. The approach we take is based on trial division. While there are much more sophisticated factorisation algorithms available, they are not as straightforward, resulting in models much more difficult to follow and evaluate. We are also not primarily interested in evaluating the efficiency of the algorithm *per se*, but rather that of the scheduling method.

At the core of the model are the factorisation events *process1*, *process2*, etc., up till the number of computational slave processes. This typically corresponds to the number of hardware computational nodes (processors or cores) to be used for slave computations. The Event-B notation of the factorisation events, in a model designed for two computational processes, is given in Figure 1. Note that we use separate events instead of parametrisation, since we want the factorisation events to be separate from each other. It was also of utmost importance that the model be

```
process1 ≜
when
    $continue\_1 > 0$
    $result\_1 \neq 0$
    $i\_1 < n/2$
then
    $result\_1 := n \bmod i\_1$
    $i\_1 := i\_1 + STEP$
    $continue\_1 := continue\_1 - 1$
end
```

```
process2 ≜
when
    $continue\_2 > 0$
    $result\_2 \neq 0$
    $i\_2 < n/2$
then
    $result\_2 := n \bmod i\_2$
    $i\_2 := i\_2 + STEP$
    $continue\_2 := continue\_2 - 1$
end
```

Figure 1: Factorisation events for two computational processes.

expressed in such a way that the factorisation events have no variables in common, since the scheduler would otherwise be unable to run them in parallel. They may, nevertheless, refer to the same constants.

There are variables i_1, i_2, etc., associated with the respective factorisation events. Variable i_1 is initialised to the value 2 (i.e. 1+1), i_2 to the value 3 (i.e. 2+1), etc., and each time a factorisation event m is executed, it checks whether the constant n is divisible by the current value of its associated variable i_m. If that is the case, a factor has been found. To distribute the work evenly among the processes, i_m is after each trial division incremented by a constant *STEP*, containing the number of factorisation events in the model. In addition to the variable i_m, each factorisation event m is also associated with a counter $continue\_m$. Initially set according to a constant *CONTINUES*, it is decreased by 1 after every trial division. By checking that $continue\_m > 0$ as part of the guard, the number of consecutive executions of each factorisation event is limited to *CONTINUES*.

Since the factorisation events must not have any variables in common, they cannot directly check whether another event has found a factor. This is where a synchronisation event *newround* comes into play. After the factorisation events have been executed for a maximum of *CONTINUES* times, they disable themselves, and can only be re-enabled by *newround*, provided that none of them has already found a factor. The listing for *newround* is given to the left in Figure 2. Note that *newround* is disabled if the value of all variables i_m is greater than $n/2$. Each of the m factorisation events also disables itself if the corresponding i_m exceeds $n/2$. This is because a factor (less than n itself) cannot exist beyond this threshold. It would actually be enough to check numbers up till $\sqrt{n}$, but since Event-B does not support square root, we use $n/2$ as the limit.

In the case that no factorisation event finds a factor, and all i_m exceed $n/2$, event *found0* becomes enabled. This event is shown to the right in Figure 2, and it simply sets a variable *result*, storing the final result, to n. There are also events *found1*, *found2*, etc., related to the factorisation events *process1*, *process2*, etc., respectively. These events, as shown in Figure 3, set the *result* variable to the

```
newround ≜
when
    $result\_1 \neq 0 \land result\_2 \neq 0$
    $\neg(i\_1 > n/2 \land i\_2 > n/2)$
    $continue\_1 < CONTINUES$
      $\lor continue\_2 < CONTINUES$
then
    $continue\_1 := CONTINUES$
    $continue\_2 := CONTINUES$
end
```

```
found0 ≜
when
    $result\_1 \neq 0$
    $result\_2 \neq 0$
    $result = -1$
    $i\_1 > n/2$
    $i\_2 > n/2$
then
    $result := n$
end
```

Figure 2: Events for re-enabling the factorisation events (left) and for finalising when it becomes clear that the number is prime (right).

```
found1 ≜
when
    $result\_1 = 0 \land result = -1$
then
    $result := i\_1 - STEP$
end
```

```
found2 ≜
when
    $result\_2 = 0 \land result = -1$
then
    $result := i\_2 - STEP$
end
```

Figure 3: Events for finalising when process 1 (left) or process 2 (right) has found a factor.

value found by their associated factorisation events. Note that even though the final result has been found once *found0* or any of the *found1*, *found2*, etc. events has been executed, there is a possibility that one or several of the factorisation events may still be executed several times afterwards. This undesired behaviour is a side effect of the independence of events, and it is aggravated by setting the *CONTINUES* constant to a large value. The choice of value for *CONTINUES* is, however, a trade-off, since setting it to a value that is too small results in excessive synchronisation by the *newround* event.

In Figure 4, we give a sequential C++ function designed to perform factorisation similarly to the model presented above. A program based on the function is used as comparison in Section 6 when evaluating the efficiency of the parallel model. Though designed to resemble as closely as possible a sequential version of the algorithm above, there are a number of differences. For example, since the program is sequential, it obviously contains no synchronisation or other process-related mechanisms, resulting in much simpler code. The sequential version also always finds the lowest factor greater than or equal to 2, whereas the Event-B model may find a greater factor depending on the relative progress of the processes.

```
long long factor(long long n) {
  long long i = 1;
  long long res = -1;
  while(i < n/2 && res != 0) {
    i++;
    res = n % i;
  }
  if(res == 0) return i; else return n;
}
```

Figure 4: The C++ function for sequential factorisation used as comparison.

6 Benchmarking

6.1 Approach

Performance of the scheduling approach discussed in previous sections has been evaluated by scheduling the testbed model on a multi-core/multi-processor system using different parameters. The scheduler was compiled together with the C++ translation of the model using the GNU Compiler Collection (GCC) [17] with the maximum (O3) level of optimisation. Since some parameters were part of the model and could not be changed afterwards, we technically compiled different models with minor changes from each other. To facilitate scripting for benchmarking purposes, we also slightly modified the scheduler as well as the model code to support additional parametrisation. We do not expect these changes to have disrupted test results by having any relevant impact on performance.

The system used for the test runs consists of two Xeon E5430 (2.66 GHz) processors, each of which has four computational cores, running a GNU/Linux operating system and the MPICH2 [26] implementation of MPI. While the *numerical* results will be dependent upon factors such as the clock frequency of the processors, instruction set architecture, performance of the system memory, etc., we believe that the *interpretation* of the results is representative of modern computer systems with similar topology (e.g. the same number of processor cores). This is because we are mainly interested in the overall feasibility of the scheduling framework and the impact of different parameter values. Since all our test runs, including the comparison with a sequential program, have been done on the same system, the results are mutually comparable to each other.

6.2 Parameters and Results

From the perspective of the scheduling platform, there are especially two parameters of interest: the number of slave processes and the value of *REPEAT* used in

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	13.37	13.36	13.36	13.36	13.36	13.36	13.36	13.36
Par. $c = 10^2$	91.42	85.52	90.45	92.00	85.37	91.34	87.43	89.08
Par. $c = 10^3$	16.57	16.59	16.78	16.63	16.94	16.55	16.07	16.59
Par. $c = 10^4$	10.23	10.28	9.92	10.17	9.57	10.18	10.26	10.09
Par. $c = 10^5$	9.29	8.90	9.31	9.55	9.38	8.68	9.60	9.24
Par. $c = 10^6$	9.19	9.46	8.13	9.13	8.82	9.19	9.24	9.02
Par. $c = 10^7$	9.31	8.47	9.44	9.15	9.18	9.32	9.33	9.17
Par. $c = 10^8$	9.26	9.48	9.47	9.46	9.49	9.58	9.38	9.45
Par. $c = 10^9$	9.51	9.44	9.54	8.70	9.46	9.51	9.30	9.35

Table 1: Test runs with 3+1 processes, n = 2,147,483,647.

the scheduler. Important parameters related to the model are n, i.e. the number to factorise, and the value of the constant *CONTINUES*. Even though we will not mention it explicitly from now on, the number of slave processes also has implications on the model in that the number of factorisation events has to match, and the value of *STEP* must be set accordingly. Furthermore, we decided to keep the values of *REPEAT* and *CONTINUES* bound to each other, even though it would not absolutely have to be that way. We motivate our decision as follows. The value of *REPEAT*, being a property of the scheduler, may have an impact on the performance of execution, but it does not change the logics of the model. In contrast, *CONTINUES* is part of the model, which is nevertheless constructed to produce a correct result for different values of *CONTINUES*. A value of *REPEAT* less than *CONTINUES* would imply that there may be unnecessary involvement of the scheduler even in cases where the slave processes could have been repeatedly executed events on their own. Since the model is not aware of the impact of the repetition mechanism of the scheduler, though interrupted, it would not even have a chance of synchronising by executing the *newround* event. A *REPEAT* value greater than *CONTINUES* is also not motivated, since repeated execution of the factorisation events would be limited by *CONTINUES* anyway.

For each set of parameters, we performed eight timed test runs. The initial one was disregarded, since it may not be comparable should subsequent executions have any caching benefits. The timings of the subsequent seven executions (numbered 1-7) were recorded, and the mean value was computed. The time unit used was seconds and fractions thereof. Our first set of runs was performed with the parameter n = 2,147,483,647 with three slave processes. An additional process was used for the scheduler, so technically, the execution involved four processes. Note that we chose n to be a prime number in order to achieve benchmarking times long enough to draw conclusions. We ran several subsequent test sets, with the values of c = *REPEAT* = *CONTINUES* being 10^2, 10^3, ..., 10^9, respectively. The results are shown in Table 1.

As can be seen in Table 1, with the c value set to 100 (i.e. 10^2), the execution

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	498.63	427.11	549.03	448.51	555.63	567.38	516.91	509.03
Par. $c = 10^2$	2844.53	2883.50	2850.31	2802.62	2846.89	2840.90	2864.31	2847.58
Par. $c = 10^3$	544.07	555.74	542.31	560.58	557.56	552.23	550.53	551.86
Par. $c = 10^4$	320.24	320.19	319.80	320.67	320.69	317.67	318.21	319.64
Par. $c = 10^5$	292.76	293.95	293.61	293.13	294.09	292.09	292.34	293.14
Par. $c = 10^6$	288.50	290.00	290.34	288.24	290.16	290.23	288.50	289.42
Par. $c = 10^7$	288.32	286.88	288.05	288.52	289.86	296.57	288.13	289.48
Par. $c = 10^8$	289.03	287.51	289.40	288.18	287.32	286.79	287.81	288.01
Par. $c = 10^9$	288.06	288.29	287.24	288.24	287.97	288.34	287.68	287.97

Table 2: Test runs with 3+1 processes, n = 68,720,001,023.

times are several times higher than that of the sequential program with a mean value of 13.36 seconds for the sequential version versus 89.08 seconds for the parallel one. It can be explained by overhead that, in this case, is clearly not outweighed by the potential benefits of parallelism. This is apparently the case even though the slave processes may allow the factorisation events to be executed up to 100 times without involving the scheduling process. The overhead may in part be due to MPI communication, but also behaviour specific to the parallel model, such as the *newround* event, may have an impact. However, if c is set to 1000, timings approach those of the sequential model, and with a c value of 10000, the parallel model is faster at 10.09 seconds on average. Values of c beyond 10^5 do not seem to provide further gains, and execution times level out at about 9 to 9.5 seconds, which constitutes approximately 70% of the running time of the sequential version. However, we also realise that execution times of only a few seconds may not necessarily be representative of performance in general. For example, the time taken to initialise the scheduling platform may have an unduly large impact. Therefore, we performed a new set of test runs with the same parameters, except for setting the value of n to 68,720,001,023, which is also a prime number. We present the results in Table 2.

The general pattern turned out to be the same as for the lower value of n. For a c value of 100, execution times are poor in this case, as well, but from c = 10000 and beyond, we see performance gains. While they also level out for higher values of c, execution times are around 50%-60% as compared to the corresponding sequential program. This is better than in the previous case. However, we were also interested in testing how the framework scales when the number of processes increases. Therefore, we did yet another set of test runs. We kept the value of n at 68,720,001,023, but increased the number of slave processes to six, in addition to the scheduling process, which is always present. The results are given in Table 3. Note that the sequential test runs used for comparison were not redone, since the value of n remained unchanged.

While we see the same pattern as before, execution times are considerably

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Mean
Sequential	498.63	427.11	549.03	448.51	555.63	567.38	516.91	509.03
Par. $c = 10^2$	2574.73	2074.18	2609.96	2647.20	2494.59	2577.47	2632.28	2515.77
Par. $c = 10^3$	338.11	319.87	347.55	335.40	324.07	348.34	346.58	337.13
Par. $c = 10^4$	159.96	137.46	165.59	141.58	160.85	158.15	153.70	153.90
Par. $c = 10^5$	147.62	146.77	147.72	147.02	121.49	148.72	146.45	143.68
Par. $c = 10^6$	113.44	144.23	136.24	145.56	145.35	145.51	145.68	139.43
Par. $c = 10^7$	145.03	145.50	134.56	146.20	129.59	145.29	146.63	141.83
Par. $c = 10^8$	119.44	146.29	144.89	134.04	145.75	139.15	130.20	137.11
Par. $c = 10^9$	140.61	120.94	139.71	138.91	140.97	142.09	141.35	137.80

Table 3: Test runs with 6+1 processes, n = 68,720,001,023.

shorter. The scenario where c = 100 is still highly inefficient, but it is nonetheless slightly faster than with three slave processes. We also note that for a c value of 1000, performance is now better than for the sequential comparison, whereas it was a bit slower than sequential in the 3+1 set-up. At c = 10000, and especially from c = 10^5, where the levelling out seems to start, performance is greatly increased as compared to using three slave processes. For such values of c, execution times in the 6+1 process set-up are around half of those in the 3+1 setting, indicating a good scalability of the scheduling approach.

7 Conclusions

In this paper, we have performed an empirical study on the efficiency of MPI-based parallel scheduling of compute-intensive code translated from an Event-B model. The purpose was to evaluate whether an on-the-fly scheduling approach taken is feasible from a practical perspective. We used an integer factorisation model as a testbed for the study. The main pitfall we suspected in the basic form of the framework was that the overhead of the scheduler and the MPI library communication would defeat the potential speed gains of parallelism. This is because individual events in Event-B are typically very fine-grained.

In an attempt to mitigate excessive overhead, we introduced an optimisation in the form of repeated event execution without the involvement of the scheduler. A benefit of this solution is that it directly reduces the communication overhead. The repetitive behaviour introduced is compatible with the original behavioural semantics typically used in Event-B, and can therefore be considered correct from a theoretical point of view. To benefit from this strategy, the model should be designed so that computational events are enabled a large number of times in a row.

We performed a number of test runs on a multi-core/multi-processor system to evaluate the performance of the testbed factorisation model when using the optimisation. The tests involved different numbers of processor cores in use, and different limits on how many times events can be executed consecutively without involving the scheduling process. The runs showed that given a large enough number of repetitions, the performance increased to a degree where the program clearly benefits from parallel execution, as compared to a corresponding sequential program. We also found that when increasing the cores in use from 3 slave processes + 1 scheduler, to a 6+1 configuration, performance increased considerably. This indicates a good scalability of the approach. In conclusion, the empirical study we have performed hints at a potential practical applicability of the run-time scheduling framework in question.

Acknowledgements

This research was supported by the EU funded FP7 project DEPLOY (214158). `http://www.deploy-project.eu`

References

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

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

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer Berlin / Heidelberg, 2006.

[5] R.J.R. Back. Refinement calculus, part II: Parallel and reactive programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer Berlin / Heidelberg, 1990.

[6] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *Proc. of the 2nd ACM SIGACTS-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[7] R.J.R. Back and J. von Wright. Refinement calculus, part I: Sequential non-deterministic programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer Berlin / Heidelberg, 1990.

[8] P. Boström. Creating sequential programs from Event-B models. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM 2010)*, pages 74–88. Springer-Verlag, 2010.

[9] P. Boström, F. Degerlund, K. Sere, and M Waldén. Concurrent scheduling of Event-B models. In *Proceedings 15th International Refinement Workshop*, pages 166–182, June 2011.

[10] ClearSy. Atelier B web site. `http://www.atelierb.eu/` .

[11] F. Degerlund. Scheduling of compute-intensive code generated from Event-B models: An empirical efficiency study. In K. Göschka and S. Haridi, editors, *Proceedings of Distributed Applications and Interoperable Systems (DAIS) 2012*, volume 7272 of *Lecture Notes in Computer Science*, pages 177–184. Springer Berlin / Heidelberg, 2012.

[12] F. Degerlund, R. Grönblom, and K. Sere. Code generation and scheduling of Event-B models. Technical Report 1027, Turku Centre for Computer Science (TUCS), 2011.

[13] F. Degerlund, M. Waldén, and K. Sere. Implementation issues concerning the action systems formalism. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'07)*, pages 471–479. IEEE Computer Society Press, 2007.

[14] Eclipse platform. `http://www.eclipse.org/` .

[15] A. Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, 2010.

[16] A. Edmunds and M. Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES 2011*, 2011.

[17] GNU Compiler Collection (GCC) web site. `http://gcc.gnu.org/` .

[18] R. Grönblom. A framework for code generation and parallel execution of Event-B models. Master's thesis, Åbo Akademi University, 2009.

[19] S. Hallerstede. Structured Event-B models and proofs. In *Abstract State Machines, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 273–286. Springer-Verlag, 2010.

[20] S. Hallerstede. On the purpose of Event-B proof obligations. *Formal Aspects of Computing*, 23:133–150, January 2011.

[21] A. Iliasov. On Event-B and control flow. Technical Report CS-TR-1159, School of Computing Science, Newcastle University, 2009.

[22] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal of Software Tools for Technology Transfer (STTT)*, 10:185–203, February 2008.

[23] D. Méry and N. K. Singh. Automatic code generation from Event-B models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188. ACM, 2011.

[24] Message Passing Interface Forum. `http://www.mpi-forum.org/` .

[25] C. Métayer. AnimB web site. `http://www.animb.org/` .

[26] MPICH2 web site. `http://www.mcs.anl.gov/research/projects/mpich2/` .

[27] Rodin platform web site. `http://www.event-b.org/` .

[28] K. Sere. *Stepwise Derivation of Parallel Algorithms*. PhD thesis, Åbo Akademi University, 1990.

[29] T. Servat. BRAMA: A new graphic animation tool for B models. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 274–276. Springer Berlin / Heidelberg, 2006.

[30] M. Waldén and K. Sere. Reasoning about Action Systems using the B-Method. *Formal Methods in System Design*, 13:5–35, May 1998.

[31] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14:221–227, April 1971.

[32] S. Wright. Using EventB to create a virtual machine instruction set architecture. *Abstract State Machines, B and Z*, pages 265–279, 2008.

Paper V

Derivation of concurrent programs by stepwise scheduling of Event-B models

Pontus Boström, Fredrik Degerlund, Kaisa Sere and Marina Waldén

Full original reference:

P. Boström, F. Degerlund, K. Sere and M. Waldén. “Derivation of concurrent programs by stepwise scheduling of Event-B models”. In: *Formal Aspects of Computing* (2012). DOI: `10.1007/s00165-012-0260-5`.

The paper is based on the following work:

P. Boström, F. Degerlund, K. Sere and M. Waldén. “Concurrent Scheduling of Event-B Models”. In: *Proceedings 15th International Refinement Workshop.* Ed. by J. Derrick, E. A. Boiten and S. Reeves. Vol. 55. Electronic Proceedings in Theoretical Computer Science (EPTCS). Open Publishing Association, 2011, pp. 166–182.

N.B. This paper is not included in the electronic edition of the thesis.

ISBN 978-952-12-2811-7
ISSN 1239-1883

Fredrik Degerlund

Scheduling of Guarded Command Based Models

Fredrik Degerlund

Scheduling of Guarded Command Based Models