Verificación de autómatas temporizados en arquitecturas monoprocesador y multiprocesador

Schapachnik, Fernando

2007

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Verificación de autómatas temporizados en arquitecturas monoprocesador y multiprocesador

Tesis presentada para optar al título de
Doctor de la Universidad de Buenos Aires
en el área Ciencias de la Computación

Fernando Schapachnik

Directores de tesis: Dr. Víctor Braberman
Dr. Alfredo Olivero

Buenos Aires, octubre de 2007

Contents

Resumen

Verificación de autómatas temporizados en arquitecturas monoprocesador y multiprocesador

Los sistemas de tiempo real están presentes en dispositivos embebidos, teléfonos celulares, controladores de vuelo, etc. Su complejidad es cada vez mayor, y cada vez cumplen funciones más críticas, donde las consecuencias de sus fallas son cada vez más graves. Por estos motivos tiene sentido realizar un análisis riguroso sobre ellos, que permita asegurar que sus diseños cumplen ciertas propiedades deseables. A este tipo de análisis se le suele llamar *verificación automática* o *model checking*.

Un formalismo muy difundido para realizar esta tarea es el de los *autómatas temporizados*, una extensión de la teoría clásica de autómatas que permite trabajar con tiempo *denso*. Si bien las técnicas basadas en este tipo de autómatas se conocen desde hace un par de décadas, sufren aún de problemas de escalabilidad, lo que dificulta una utilización mayor en casos reales.

Esta tesis analiza diversos mecanismos para acelerar la verificación de autómatas temporizados, tanto en el (clásico) ambiente monoprocesador, como así también en arquitecturas distribuidas. Durante el transcurso de la misma se construyó el *model checker* Zeus, que se utiliza para la experimentación.

Luego de presentar la motivación e introducción al tema, se presentan los conceptos básicos del formalismo y las estructuras de datos más utilizadas en este tipo de herramientas. Se destacan los dos algoritmos tradicionales de verificación, *forward* y *backwards*.

A continuación, se explora la paralelización del algoritmo *backwards*: se presenta una prueba de corrección de una versión distribuida y asincrónica del mismo, se la implementa, y se experimenta también con una versión sincrónica y con otra que reparte la carga de trabajo sobre los procesadores de manera dinámica. Se analizan los resultados obtenidos y se argumenta que son cercanos a óptimos utilizando la unidad de distribución de trabajo actual: la asignación de nodos del autómata a procesadores.

Luego se aborda la paralelización del algoritmo *forward* junto con una estrategia para la redistribución dinámica de trabajo entre los procesadores. Después de demostrar su corrección, se analizan varios casos de estudio sobre distintas configuraciones, obteniéndose resultados positivos pero no óptimos, y comprobándose que aún queda mucho por investigar con respecto a la distribución de este algoritmo, y planteándose varias posibles líneas de investigación.

En el capítulo siguiente se compara el presente trabajo con otros del área, y se argumenta que ciertos resultados positivos obtenidos por otros investigadores tienen como causa no la paralelización en sí, sino ciertos efectos colaterales de la misma, que tal vez podrían reproducirse en un monoprocesador.

A continuación se presenta trabajo realizado en pos de obtener una nueva estructura de datos. En ese marco, se introducen los rCDDs (*restricted Clock Difference Diagrams*) junto con casos de estudio en los que estos mejoran los tiempos de ejecución hasta en un treinta por ciento.

El siguiente capítulo explora dos optimizaciones también válidas para el caso monoprocesador. La primera consiste en computar rangos posibles para los valores de las estructuras de datos y de esa manera reacomodarlas para detectar antes ciertas condiciones de corte de operaciones utilizadas con

mucha frecuencia. Los resultados muestran mejoras de hasta un diecisiete por ciento en los tiempos de ejecución. La segunda consiste en computar una aproximación al hipervolumen de los poliedros representados por las estructuras de datos, de manera tal de evitar $O(n^2)$ comparaciones al costo de $O(1)$. Esta técnica logra una reducción de tiempo de hasta casi un veinte por ciento.

El anteúltimo capítulo presenta VInTiMe, una herramienta gráfica que permite describir diseños de tiempo real, especificar sus propiedades de manera amigable, y verificarlas utilizando el *model checker* distribuido Zeus, de manera sencilla. Constituye un esfuerzo por acercar los métodos formales al usuario no tan experimentado.

Finalmente, se exponen las conclusiones del trabajo, donde se argumenta la dificultad de decidir de antemano qué optimizaciones y/o parámetros serán más convenientes para un caso de estudio dado, y se proponen ideas alternativas para atacar el problema.

Palabras clave: Autómatas temporizados, verificación automática, sistemas de tiempo real, computación distribuida, computación paralela, multiprocesador, Zeus.

Summary

Timed Automata Model Checking in Monoprocessor and Multiprocessor Architectures

Real-time systems are present today in embedded devices, cellphones, flight controllers, etc. Their complexity is ever increasing, as well as the criticality of their functions. Consequences of failure are each day more dangerous. This is why performing a rigorous analysis over them makes sense, allowing to assert that their design complies with some desirable properties. This type of analysis is usually called *automated verification* or *model checking*.

A well-known formalism for model checking is *timed automata*, an extension of the classical automata theory allowing to model *dense* time. Although timed automata-based techniques are known since a couple of decades ago, they still suffer from scalability issues, jeopardizing a broader adoption in real cases.

This thesis analyzes diverse mechanisms to speedup timed automata verification, both in the (classical) monoprocessor environment, as well as in distributed architectures. While working on it, the model checker Zeus was built as a base for experimentation.

After motivating and introducing the subject matter, foundational concepts of the formalism are presented, as well as the data structures more used by the tools. Special attention is payed to the two traditional verification algorithms, *forward* and *backwards*.

Next, the parallelization of the *backwards* algorithm is explored: a correctness proof of a distributed and asynchronous version is presented. It is implemented, and experimentation is also done with a synchronous version and another one that dynamically balances workload among processors. Obtained results are analyzed and it is argued that they are close to optimal, as long as the same unit of distribution is used: assigning each automata node to only one processor.

After that, a distributed version of the *forward* algorithm is analyzed, along with a dynamic workload migration strategy. Its correctness is proved, and various case studies over different configuration settings are tried. Positive results are achieved, yet not optimal, corroborating that there is still much research to perform regarding this algorithm. Also, various possible future continuation lines are presented.

Next chapter compares against other work in the literature, and argues than certain positive results achieved by other researchers might be caused not by the distribution *per se*, but actually as a byproduct of it, and that the same effects might be reproducible in a monoprocessor setting.

Work towards a new data structure is next, introducing rCDDs (short for *restricted Clock Difference Diagrams*). Along with their details and algorithms, case studies showing up to a thirty percent of improvement in running times are presented.

After that, two optimizations also valid for the monoprocessor setting are introduced. The first one computes ranges of possible values for the data structures, trying to accommodate them in such a way that certain finishing conditions for the most used operation are met earlier. Results show improvements of up to a seventeen percent in running times. The second one is about computing an

approximation to the hypervolume of the polyhedra represented by the data structures, thus avoiding $O(n^2)$ comparisons at the price of $O(1)$. This technique allows to reduce running times by up to almost twenty percent.

The penultimate chapter presents VInTiMe, a graphical tool that allows to describe real-time system designs, specify their properties in a friendly way, and verify them using the distributed model checker Zeus, in a simple manner. This is an efford to put formal methods closer to the practitioner.

Lastly, conclusions are presented, arguing the difficulty of deciding beforehand which optimizations and parameters would be better for a particular case study, and proposing alternative ideas to palliate such an issue.

Keywords: Timed Automata, Model Checking, Automated Verification, Real-Time Systems, Distributed Computing, Parallel Computing, Multiprocessor, Zeus.

Acknowledgements

Double mention goes to my family. On one hand, this thesis stole time from them. On the other, my parents Paulina and Edgardo not only supported my education, but also encouraged me to undertake it in different ways during my early days. I hope I can pay a little more attention to them, my brothers Alejo and Rodrigo, and the rest of the family.

University of Buenos Aires received me at my twelve, and since then was charged with the responsibility of educating me. I don't have enough words to thank for what I received, from both the University and their staff, and hope that my teaching there serves to give the same to others. University of Buenos Aires is a free, secular, jointly run by students, former students and teachers, public institution, and I'm very proud of all that.

I'd would like to thank the following people, who worked directly in projects closely related to this thesis:

- Jorge Lucángeli Obes and Pablo Rodríguez Zívic who, with their hacking, brightly squashed many bugs from Zeus, and also implemented a couple of features.
- Lucía Cavatorta, Andrés Ferrari and Guido de Caso, performed and excellent job programming VInTiMe, and gluing Zeus into it.
- Alejandra Alfonso, Diego Garbervetsky and Nicolás Kicillof, who worked on some of the other tools that compose VInTiMe.
- Esteban Pavese, who did an amazing job on rCDDs.

I'd like to thank the people at IBM Eclipse Innovation Grant, who believed in our ideas and thus founded part of VInTiMe and related projects.

I would also like to thank Sergio Yovine, who years ago allowed us to get access to Kronos' source code, Esteban Mocskos, who undertook most of the work of our shared responsibility as IT Managers in the Department, and Aldo Rosenberg, my boss at the Ministry of Finance who allowed me to work part-time. My gratitude to the those who bear with me every day: my students, my coworkers at the University (so many along the years to mention!), and my coworkers at the Ministry, specially to the closest ones: Bruno, Carolina, Celeste, Damián, Florencia, Gabriela, and Tomás.

Special mention goes to Alfredo and Víctor. They are great people and advisors, and working with them is a pleasure for me. Their guidance has always improved things.

In his PhD thesis, my friend Santiago Figueira mentioned a bunch of people to whom he thanked for the inspiring and funny environment our Department is. I can only adhere to his thoughts, but will spare the names, being afraid of leaving somebody out! For the record, being surrounded by so many extremely intelligent individuals is one of the most pleasurables experiences of my everyday life. Not a day goes by without somebody casually telling you a completely clever and non-trivial idea.

Finishing a thesis like this is a happy moment, but life also presented sourness. Special thanks to my close friends, who stood by me in difficult times: Diana, Carlitos, Joaquín, Martín, Patricio and Sebastián.

Chapter 1

Introduction

1.1. Foreword

A PhD thesis is a complex document, and this one is no exception: it deals with complex concepts, proposes non-trivial ideas, presents copious data as evidence, and deals with work that took many years. Also, it is expected to be read by different audiences, with varying background.

This chapter aims to ease the burden of understanding the structure of the document, give a sense of "what's where" and, fundamentally, set the topic. Specifically, it is based on intuitions and assertions, and is by far the most readable by a casual bystander or the curious beginner. As such, it should cause no surprise that it does not abound on proofs, detailed explanations or citations, that would probably conspire against the uncomplicated experience it tries to provide. The more experienced reader shouldn't be afraid. Whatever is said here will be approached again in a rigorous manner in other parts of this same document.

Let's start by outlining why distributed verification of timed systems is an interesting topic and one worth exploring.

1.2. Motivation

In current days timed systems are both pervasive and critical, ranging from embedded and PDAs to plant and flight controllers. Their complexity is ever increasing so automated ways of verifying them make sense. Automated methods, however, are known to suffer from scalability problems: their time and memory requirements grow exponentially as systems increase in size. This is why any technique that can palliate such problems is useful.

Before dealing with *what* we do with such systems, let's start with *which* systems we consider. We mentioned *timed systems*, and that is a good starting point: we deal with systems where the elapsed time between events is important, where questions like "Does the engine stop no longer that 5 ms after the stop button is pressed?" or "Does the OS awaken the thread no after 3 ms of the interruption arrival?" are relevant. In the next section we dig a little more into the semantics of our definition of time.

When we say that we deal with this kind of systems, it should be understood that we actually deal with a *description* of them, schematic enough to represent the interesting parts, while leaving out the details. This schematic description is actually a kind of automata, known as *timed automata*. They will be presented in detail in Section 2.1.

To summarize, we will focus on verification of real-time systems modeled as timed automata, which are, roughly speaking, traditional automata with clock annotations to handle *dense real-time*.

By *verification* we actually mean a procedure known as *model checking*. Given a timed automaton $\mathcal{A}$ representing a system, and some (for instance, *safety*) property φ over it, does $\mathcal{A}$ comply with φ? Model checking stands for an effective procedure to answer these kind of questions. Its name comes from a modal logic view of the same problem: does $\mathcal{A} \models \varphi$?

The past few decades have seen a fair amount of effort gone into, on one hand, constructing automated tools to answer such queries, and on the other, to increasing their power so bigger and more complex systems can go through the verification process.

Still, scalability is a major obstacle for a wider adoption of model checking technology. Verifying even medium-sized designs can quickly exhaust memory or processing capacity of rather powerful computers. This is partly due to the *state explosion* problem: small increases in system complexities generate a huge number of new possible system states, all of which must be explored. Same thing said from a complexity-theoretical stand: model checking the kind of systems we are interested in is a PSPACE-complete problem.

In recent years, there has been an increasing interest in the use of Distributed Computing as a way to augment the size of the models tools can deal with.

This thesis presents research around distributed model checking of timed automata. Its main question is whether practical improvements could be made to the verification of timed automata by working with more than one processor. While building a distributed tool was the objective, some improvements, data structures and conclusions valid also for the monoprocessor case have been found.

All the effort has gone into diminishing wall-clock time. The measure of success is time saved per extra computational resource invested.

Before going on is better to address an important question: when we refer to *timed systems*, what do we actually mean?

1.3. Time Models

What does *dense real-time* mean? Time can be modelled by very different theoretical formulations[1], each one having advantages and disadvantages over the other. What's most important, is that there is no model that is clearly superior or fit for any circumstance. We will first describe the most common approaches and then discuss which one are better suited for our scenarios of interest.

- The *discrete-time* model requires the time sequence to be a monotonically increasing sequence of integers. Certain types of application domains can make such assumptions, notably synchronous digital circuits, where signal changes are considered to happen exactly when a *tick* signal arrives. Although many types of behaviours can not be captured in this model, its principal advantage is that simple variations of traditional finite automata theory can be used to deal with them.
- A variation of the former, the *fictitious-clock* model, requires only the sequence of integer times to be non-decreasing. The interpretation is that events occur at real-times instants, but only the integer part of the clock reading is considered. Its approximate nature being obvious, it also can be manipulated via simple variations of traditional finite automata theory.
- In the *dense-time* model, time is thought of as a dense set. In this model the times of events are real numbers which increase monotonically without bound. Physical events comfortably fit in this model.

If the only important aspect of a system is whether event A happens before or after event B, any of two first models is an appropriate choice for expressing it. This type of systems, and the logics that can express properties over them[2] are called *temporal*. Temporal analysis is very useful for some

[1] We follow [AD94] for this presentation.
[2] Such as *LTL*, the *Linear Temporal Logic*.

scenarios. For instance, checking whether before each granting of access to a secured object there is always a password check. There are many successful tools in this area, such as SPIN [Hol03] or SMV [BCM+92], among others.

Consider, for example, the following requirement: the emergency brake should be functional after no more than five seconds after pressing the stop button. Temporal analysis is not enough to express this: it can only determine if the brake would engage after pressing the button, but not how far apart. If precise time bounds are required we talk about *timed systems* (as opposed to temporal).

It should be noted that in timed systems, the discrete-time or fictitious-clock models are usually insufficient, as they cannot be used to express that, no matter how near event B happens after event A, event C can occur in between them, a frequent necessity when dealing with physical systems.

Also, there are systems with no explicit notion of time. By extension, we will call *untimed* both to them and the temporal ones. This thesis focuses on the verification of dense real-time timed systems. More specifically, the ones that are modeled as timed automata.

1.4. Model Checking

We have already mentioned that our interest lays in model checking of timed automata, and that by that we mean effective procedures to decide if timed systems comply or not with certain properties.

But why is this hard? Although next chapter answers with precision, some intuition could actually help.

Timed automata can be thought of as regular finite automata with the addition of clocks. Let's forget about the clocks for a moment. Also, let's consider only the problem of checking if it has some forbidden behaviour. I.e., the problem of checking whether some word is or not in the automata language. In turn, it means traversing the complete graph underlying the automata. To simplify things, let's assume the language is prefix-free. On the worst case it takes $O(n.m)$ for a graph with n nodes and m edges.

If our system has k components, each with n_i and m_i nodes and edges, then it parallel composition can be, on the worst case, of a size similar to the product automaton: $O(\Pi n_i)$ nodes and $O(\Pi m_i)$ edges, leading to a traversal complexity that grows geometrically with each new components. Adding the clocks means, in principle, extra work: checking clock restrictions in each vertex.

The aim of the preceding paragraphs is giving an idea of why model checking is such a hard problem. Although many optimizations have been devised for it, it still makes sense to put more than one computer to work on it at the same time. This approach is called distributed computing and next section deals with it.

1.5. Challenges of Distributed Computing

Although the idea of putting many processors to work in the same problem seems simple at first, a more detailed analysis is required. Some of the perspectives to consider include:

- Architecture: multiple processors on shared memory are generally easy to communicate but require locking of data structures and appropriate, ad-hoc hardware. On distributed algorithms, on the other hand, communication methods (such as message passing) must be carefully chosen. Also synchronization is usually a hard problem. On the plus side, powerful clusters can be built from commodity hardware. This dichotomy is generally referred as *parallel vs. distributed* computing.

- Overhead: clearly, distributing the problem and then combining the results involves, by itself, a number of computational resources. Not investing enough of them can cause load balancing problems, but an excess is also not wanted. A careful equilibrium has to be achieved, trying to avoid an important overhead.
- Orchestration: having multiple actors (processors in this case) working together very often requires some kind of coordination among them. How should this coordination be shaped? In this aspect the tension is between centralized vs. distributed control.
- Distributability of the problem: some problems are naturally more easy to distribute than others (think, for instance, in integer matrix multiplication). The worst case presents when there are important interdepencies between the results computed by one processor and the others. Typical cases of bad conditioned problems include fixed points and some kind of graph problems where edges has to be followed to obtain data about neighbour nodes. Unluckily, the problem dealt here combines both.

The usefulness of a distributed strategy is often measured by the *speedup* gained. Speedup with n processors is computed as $\frac{t_1}{t_n}$ where t_i is the time it takes to finish the verification with i processors. The goal is usually to get (close to) linear speedups, although verifying cases where the monoprocessor version exhausted its memory is also considered a success.

Some mention is deserved to the notorious Amdahl's [Amd67] and Gustafson's [Gus88] Laws, which were believed to be different until Shi [Shi96] cleared the confusion. This law separates the serial and the parallel part of a program, and states speedup in terms of only the last one. However, as also mentioned by Shi, its usage is mostly theoretical as it contains many factors that can only be obtained through experimentation, if they can be calculated at all.

The rest of the thesis, specifically the Chapters 4 and 5 can be considered efforts towards better speedups.

1.6. Zeus, the tool

Given that our measure of success is speedup, an empirical measure, we actually needed a tool to program all our strategies and see how they performed. In our case, this tool was the model checker ZEUS.

Building its approximately 30000 lines of C code was a journey into knowledge. There's no better way to understand an algorithm than having to code it, even your own's!

ZEUS was conceived as a testbed for experimenting ideas, and thus was developed following a software architecture centric approach, where separation of concerns as a way to achieve modularity was a key design decision.

It's first version not only conceptually descended from the KRONOS tool [BDM+98][3], but also used its engine for the main data structures and was used for the work in Chapter 4, being the first distributed one of its kind. The current one was remade from scratch and served for the rest of the research. We believe it to be the only distributed model checker for timed automata that is freely available (see Chapter 9).

It was integrated in the VINTIME verification framework –described in Chapter 9– which can be

[3] According to Greek mythology Zeus was the son of Rhea and Kronos, the supreme god. After its birth Rhea hid Zeus from his father, fearing that he would destroy his own descendants, as his fate was to be replaced by one of them. To deceive him, she handed Kronos a clothed rock, that he immediately devoured. Latter, Zeus plotted a rebellion with his brothers and the Olympics defeated the Titans in battle. The myth also tells that years after Zeus seduced Metis, titaness that reigned over knowledge and wisdom, only to discover that the lineage that she would gave birth to where destined to replace him, so he eat her alive. Interestingly enough, early version of our ZEUS tools where linked to the METIS library.

freely downloaded on-line.

Next section walks the reader through the structure of the rest of thesis, as an aid for its reading.

1.7. Road map

Next chapter gives an introduction to the terminology used through the rest of the thesis, and some background on timed automata verification. It does not aim to be a complete presentation of the topic. Instead it showcases the most significant notions using formal definitions when required for a better understanding, but resorting to informal explanations (and corresponding citations for a full presentation) if possible.

As data structures constitute an important part of the work presented here, Chapter 3 is devoted to them. It presents in some detail the most commonly used ones in the literature and the tools. Although matrices[4] dominate the scene, some work has been done on decision tree-like structures.

Our approach to timed automata model checking is based on reachability: is there any path from the initial state to some distinguished (target) states?

Two main algorithms exist to verify reachability: one starts its traversal by the target states and moves by predecessors trying to reach the initial ones, and is called backwards. Forward, on the other hand, tries to find a way through successors from the initial to the target states. Some more details about them is presented in Section 2.7.

Our attempt to distribute backwards is described in Chapter 4. We believe we have found some "end of the road" signs for this effort, yet have beaten some important obstacles and made interesting contributions in the area.

Forward is dealt with in Chapter 5. It features an in-depth analysis of different workload balancing settings, showing interesting results and future ways of improving them. Contrasting the previous chapter, it shows how much work there is still to do in the area.

Much successful work has been done to distribute *untimed* model checkers, but little in the timed distributed realm. Chapter 6 surveys the state of the art and compares our work with the few other similar ones.

After that, Chapter 7 revisits data structures to propose an improvement over tree-like representations.

While working on a distributed model checking tool some interesting improvements have been found for the monoprocessor case also, which are described in Chapter 8.

Chapter 9 describes VInTiMe, the verification framework into which our model checker has been integrated.

Lastly, Chapter 10 deals with conclusions and perspective, and Appendix A describes the case studies used.

1.8. Summary of Contributions

The original contributions of this thesis are:

- A thorough exploration of distribution strategies for timed automata reachability algorithms (Chapters 4 and 5). Not only non-naïve distributed algorithms are proposed and implemented, but also correctness proofs are provided for them.

[4] Not numerical ones.

- Exposing the limits of the backwards algorithm in regards to its distributability, as well as proposing palliating strategies to overcome those (see Section 4.6). Parallelizing the backwards algorithm is an area we pioneered (cf. [SBO02, BOS02]).
- Presenting complex load balancing strategies for both the backwards (Section 4.5) and the forward (Section 5.3) algorithm, along with proofs of their correctness.
- Analysing in detail different workload balancing settings for the forward algorithm (Section 5.4), showing interesting results and future ways of improving them (Section 5.5).
- Presenting a detailed analysis of the results of other researchers of the field.
- Building, Zeus, a working timed automata distributed model checker, made of approx. 30000 lines of C code.
- Implementing in it the above mentioned load balancing strategies, which are far from trivial to code.
- Integrating it into the VInTiMe verification framework (see Chapter 9), which can be downloaded on-line.
- Implementing a version of the data structures that combines many of the well known abstraction techniques and a very compact representation (see Section 3.3).
- Providing empirical validation of the performance of each proposed alternative.
- Exploring alternative data structures, also valid for monoprocessor use (Chapter 7).
- Introducing hypervolume and clock reordering optimizations, useful in both the monoprocessor and distributed versions (Chapter 8).

1.9. Credits

All the work presented here (except for Chapter 7, see below) is joint work with Víctor Braberman and Alfredo Olivero, my advisors.

I'm the main author of Zeus (the tool) and all of its bugs. The backwards version of Chapter 4 used the Kronos DBM libraries, which were rewritten from scratch for the forward version.

Jorge Lucángeli Obes and Pablo Rodríguez Zívic coded some parts of the forward distributed model checker. Jorge also coded for the work in Section 8.2.

The work of Chapter 7, including all the coding, was mainly done by Esteban Pavese as his grade thesis. I was his advisor and Alfredo Olivero co-advisored.

The VInTiMe tool, presented in Chapter 9, is joint work with Lucía Cavatorta, Guido de Caso, Andrés Ferrari, Víctor Braberman, Diego Garbervetsky, Nicolás Kicillof, Alfredo Olivero and Alejandra Alfonso.

I sincerely thank all of them.

Chapter 2

Preliminaries

This chapter defines and sets notation for the concepts that will be needed to understand the rest of thesis. It does not aim to be a complete presentation of the topic. Instead it showcases the most significant notions using formal definitions when required for a better understanding, but resorting to informal explanations (and corresponding citations for a full presentation) if possible.

2.1. Timed Automata

We will focus on a given formalism to represent timed systems: timed automata. Having been introduced in [AD94][1], they have become a widely used formalism to model and analyze timed systems, and several tools support them (see Kronos [DOTY96] or UPPAAL [BLL+95], for notorious examples).

They are traditional finite automata with the addition of clocks to model (dense) time. Their semantics is based on labeled state-transition systems and time-divergent runs over them. Here we present their basic notions and refer the reader to [AD94, DOTY96, Yov97, HNSY94] –whose style we follow– for a complete formal presentation (the background provided here should be, however, enough to comprehend the rest of the thesis).

Definition 2.1 (Timed Automaton)
A timed automaton –TA *for short– is a tuple* $\mathcal{A} = \langle \mathsf{L}, X, \Sigma, E, I, \mathsf{l}_0 \rangle$*, where*

- L *is a finite set of* locations*,*
- X *is a finite set of* clocks *(non-negative real variables),*
- Σ *is a set of* labels*,*
- E *is a finite set of* edges*, and*
- $I : \mathsf{L} \xrightarrow{tot} \Psi_X$ *is a total function associating to each location a clock constraint called the* location's invariant*, and*
- $\mathsf{l}_0 \in \mathsf{L}$ *is the* initial location*.*

Each edge in E *is a tuple* $\langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}' \rangle$*, where*

- $\mathsf{l} \in \mathsf{L}$ *is the edge's source location,*
- $\mathsf{l}' \in \mathsf{L}$ *is the edge's target location,*
- $a \in \Sigma$ *is the label,*

[1] Actually, the timed automata introduced in [AD94] had Büchi acceptance conditions. Henzinger et al. [HNSY94] introduced *timed safety automata* replacing the acceptance conditions by locations invariants. The literature usually calls timed automata to the timed safety automata, and so will we.

- *$\psi \in \Psi_X$ is the guard, and*
- *$\alpha \subseteq X$ is the set of clocks that reset at the edge.*

The set of clock constraints *Ψ_X for a set of clocks X is defined according to the following grammar: $\Psi_X \ni \psi ::= x \prec c | \psi \wedge \psi | \neg\psi$, where $x \in X, \prec \in \{<, \leq\}$ and $c \in \mathbb{N}$. Ψ'_X extends Ψ_X allowing also constraints of the form $x - y \prec c$, where $x, y \in X$ and $c \in \mathbb{Z}$. We will assume invariants are backwards closed (see Definition 2.4).*

Usually, a TA $\mathcal{A}$ has an associated mapping $Pr : \mathsf{L} \mapsto 2^{Props}$ which assigns to each location a subset of propositional variables from the set *Props*.

At any time, the *state* of the system is determined by the location and the values of clocks, which must satisfy the location invariant. The precise definition of *invariant satisfaction* is given in the following two definitions:

Definition 2.2 (Clock Valuation)
Given a clock set X, a valuation *is a total function that maps clocks to non-negative real numbers. I.e., $v : X \overset{tot}{\to} \mathbb{R}_0^+$.*

A few important points about valuations:

- *Intuitively, a valuation gives the reading of each clock in a particular moment.*
- *To express the increasing of a valuation v, we write $v + \delta$ where $\delta \in \mathbb{R}$, meaning that $v + \delta$ is a new valuation such that $(v + \delta)(x) = v(x) + \delta$ for every $x \in X$.*
- *Sometimes it is useful to talk about all the possible valuations of a given clock set X. We do this by writing $\mathcal{V}_X$, the set of all the valuation functions over X.*
- *To properly define clock resets (Definition 2.6) the following notational convention will be made: whenever $\mathcal{V}_X$ is used, it should be understood as the set $\mathcal{V}_{X\cup\{0\}}$ where 0 is a special clock $(0 \notin X)$ such that $\forall v \in \mathcal{V}_{X\cup\{0\}}, v(0) = 0$ (i.e., it is a clock which always reads zero).*

Definition 2.3 (Satisfaction of a Clock Constraint)
We say that a valuation v satisfies a clock constraint ψ if the point in the multidimensional space specified by v is included in ψ. Formally, $v \in \mathcal{V}_X \models \psi \in \Psi'_X$ is defined inductively as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall x, x' \in X, c \in \mathbb{N}, d \in \mathbb{Z} :} \\
v \models x < c & \Leftrightarrow & v(x) < c \\
v \models x = c & \Leftrightarrow & v(x) = c \\
v \models c < x & \Leftrightarrow & c < v(x) \\
v \models c = x & \Leftrightarrow & c = v(x) \\
v \models x - x' < d & \Leftrightarrow & v(x) - v(x') < d \\
v \models x - x' = d & \Leftrightarrow & v(x) - v(x') = d \\
v \models \psi \wedge \psi' & \Leftrightarrow & v \models \psi' \wedge v \models \psi'' \\
v \models \neg\psi' & \Leftrightarrow & v \not\models \psi'
\end{array}
$$

Definition 2.4 (Bacwards Closed Constraint)
A clock constraint ψ is backwards closed (or past-closed) if for every valuation $v \in \mathcal{V}_X$ it holds that $v \models \psi \implies (\forall \delta \in \mathbb{R}^+)\ (v - \delta \models \psi)$.

Informally, backwards closed constraints enforce upper limits in the value of clocks.

We have now the proper background to formally define the state space of $\mathcal{A}$, by means of Definition 2.5, the definition of a state.

Definition 2.5 (State of a TA)
Given a TA $\mathcal{A}$, its state set Q is the set of tuples (l, v) where l is a location of $\mathcal{A}$ and v is a valuation satisfying the invariant of l (i.e., $v \in \mathcal{V}_X \wedge v \models I(\mathsf{l})$).

A system modeled via a TA can evolve in two different ways:

- either an enabled transition is taken, changing the location and (maybe) resetting some clocks while the others keep their values unaltered (a discrete step), or
- it may let some amount of time pass (a timed step).

In the last case the system remains in the same location and all clocks increase according to the elapsed time, while still satisfying the location invariant. Both possible evolutions will be formally defined in Definition 2.7, but we need to state first what clock reset precisely mean:

Definition 2.6 (Clock Reset)
A clock reset function α is a total function from X to $X \cup \{0\}$, which keeps some clocks unchanged while setting others to zero (i.e., such that $(\forall x \in X)\ \alpha(x) \in \{x, 0\}$).

To understand its use, let's remember that $v(0)$ is always 0. As a notational convenience, if $v \in V_X$ and α is a clock reset function, we will write $v[\alpha]$ to mean the valuation such that $v[\alpha](x) = v(\alpha(x))$ for all $x \in X$.

Having said that, let's look at the state transitions:

Definition 2.7 (Transition Relation)
The transition relation among states of Q, $\rightarrow \subseteq Q \times (E \cup \mathbb{R}^+) \times Q$ is given by the following rules:

- *Discrete transitions:*
$$\frac{\langle \mathsf{l}_i, e, \psi, \alpha, \mathsf{l}_j \rangle \in \mathcal{A} \quad v \models \psi \quad v[\alpha] \models I(\mathsf{l}_j)}{(\mathsf{l_i}, v) \xrightarrow{e} (\mathsf{l_j}, v[\alpha])}$$
  The state $(\mathsf{l_j}, v[\alpha])$ is called the discrete successor *of $(\mathsf{l_i}, v)$ and the last one the* discrete predecessor *of the former.*
- *Timed transitions:*
$$\frac{\delta \in \mathbb{R}^+ \quad (\forall \delta')(\delta' \in \mathbb{R}^+ \wedge \delta' \leq \delta \implies v + \delta' \models I(\mathsf{l}))}{(\mathsf{l}, v) \xrightarrow{\delta} (\mathsf{l}, v + \delta)}$$
  The state $(\mathsf{l}, v + \delta)$ is called the timed successor *of (l, v) and the last one the* timed predecessor *of the former.*

Intuitively:

- *Discrete transitions make the automaton evolve to another state where valuations are equal (i.e., no time has passed) except for resetted clocks. For those type of transitions to be enabled, the valuation of the original state must satisfy the transition guard ($v \models \psi$) and the resulting valuation the target state's invariant ($v[\alpha] \models I(\mathsf{l}_j)$).*
- *In timed transitions there is no location change; only time passing. Their only enabling condition is that the locations invariant should still be satisfiable.*

It should be noted that $\langle Q, \rightarrow, E \cup \mathbb{R}^+ \rangle$ constitutes a *labeled transition system* (or *LTS* for short).

2.2. Runs

Definition 2.8 (Runs)

If $\langle Q, \rightarrow, E \cup \mathbb{R}^+ \rangle$ *is the LTS of* $\mathcal{A}$*, a* run r *over* $\mathcal{A}$ *is an infinite sequence of steps through* $\rightarrow$ *such that:*

$$\begin{array}{rcl} (\forall i)(i \in \mathbb{N} & \Longrightarrow & q_i \in Q \wedge l_i \in E \cup \mathbb{R}^+) \\ r & = & q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \ldots \xrightarrow{l_{i-1}} q_i \xrightarrow{l_i} \ldots \end{array}$$

2.3. Model Checking

Model checking historically stood for a procedure for checking if some *Kripke-structure* is a model of a given modal formula. The Kripke-structure would capture the behaviour of a system or program, and the formula will be an assertion about it. By extension, model checking stands for an effective procedure to determine if a system or program (expressed in some formalism) complies to a certain behavior (given in some other formalism).

If timed automata are used as system models, it is common to use as special timed logic, called *TCTL* [ACD93], to express –fairly complex– properties over them. TCTL stands for *Timed Computational Tree Logic*, and, as implied by its name, it's a extension of the branching *CTL* logic to allow quantitative temporal operators such as $\exists\Diamond_{\leq 8}$ to mean "possibly within 8 time units". I.e., if $\mathcal{A}$ is a TA and φ is a TCTL formula, model checking means answering whether or not $\mathcal{A} \models \varphi$.

2.4. Symbolic States

It should be noted that the labeled transition system $\langle Q, \rightarrow, E \cup \mathbb{R}^+ \rangle$ is infinite because for any given state an infinite number of timed transitions can be taken. Fortunately Alur, Courcoubetis and Dill solved the problem of model checking timed automata in [ACD93] by constructing a finite quotient graph of $\langle Q, \rightarrow, E \cup \mathbb{R}^+ \rangle$, called *region graph*. This graph is finite but exponential in the number of clocks. Their result can be summarized in the following theorem.

Theorem 2.1 (TCTL Model Checking is PSPACE-Complete)
TCTL model checking over TA is PSPACE-complete.

Furthermore, its time complexity is $O(mn!2^nC^n)$ *where* m *is the number of locations,* n *is the number of clocks and* C *is the smallest natural number bigger or equal to the maximum module number appearing in any constraint of the TA.*

Proof

See [ACD93].

□

A key idea of Theorem 2.1 is the discretization of the continuous time space of clock valuations into equivalence classes. Such partitioning is based on the notion that clock valuations that either match in their integer part or are bigger than the maximum constant are essentially equivalent. That is:

Definition 2.9 (Valuations' Equivalence Relation)
Let $\widehat{\Psi} \subseteq \Psi'_X$ be a non-empty finite set of clock constraints over X. Let $C \in \mathbb{N}$ be the smallest constant that is bigger or equal $|c|$, for all $c \in \mathbb{Z}$ appearing in a clock constraint in $\widehat{\Psi}$.

We define $\simeq_{\widehat{\Psi}} \subseteq \mathcal{V}_X \times \mathcal{V}_X$ as the biggest reflexive, symmetric relation such that for all $v, v' \in \mathcal{V}_X$, $v \simeq_{\widehat{\Psi}} v'$ iff for all $x, x' \in X$:

- *If $v(x) > C$ then $v'(x) > C$.*
- *If $v(x) \leq C$ then:*
  - $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$
  - $v(x) - \lfloor v(x) \rfloor = 0 \implies v'(x) - \lfloor v(x)' \rfloor = 0$
- *For all constraint $r \in \widehat{\Psi}$, if $v \models r$, then $v' \models r$.*

We will write $[v]$ to refer to the equivalence class of v.

$\simeq_{\widehat{\Psi}}$ has the nice property of being congruent with regards to both clock reset and time passage, the time operators needed for model checking, which will be introduced below. Also if $v \simeq_{\widehat{\Psi}} v'$ and $\psi \in \widehat{\Psi}$, then $v \models \psi$ iff $v' \models \psi$.

It is not usually necessary to construct the complete region graph: maybe only a portion of it is needed for the problem at hand. More importantly, state sets can be represented symbolically, providing a coarser partition, and thus, less states. The specifics can be found in [HNSY94, Yov97]. In practice, to deal with infinite state manipulation, convex sets of clock valuations are symbolically represented as conjunctions of inequalities in Ψ'_X (e.g., $1 \leq x \leq 5 \wedge x - y > 8$):

Definition 2.10 (Zone)
A convex set of clock valuations represented as a conjunction of clock constraints is called a zone. *Non-convex sets of valuations are represented as the union of zones, and called* regions [2].

Observation 2.1 *Sometimes we will use the term* zone *to refer not only to the constraint conjunction but also to the valuation space described by them.*

Definition 2.11 (Symbolic state)
A symbolic state is a tuple (l, z) where l is a location and z is a zone.

Symbolic states are the base of implementable model checking algorithms (such as the ones described in Section 2.7). The idea is that a symbolic state graph is traversed instead of the (infinite) LTS. The symbolic state graph is one of many possible *abstractions* that have been identified in timed systems, albeit, the most important one.

2.5. Parallel Composition

The parallel composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ of TAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined using a label-synchronized product of automata [AD94, DOTY96].

To model a complex system, an automaton can be expressed as the parallel composition of the automata representing each component. A location of the obtained automaton, called *global location* or *composed automata node*, is a tuple consisting of a location of each component. Similarly, a state of the automaton (*global state*) is a global location plus the values of all clocks.

[2] Although having common roots, regions (in the sense of union of zones) and the *region graph* are independent concepts. The use of regions (in the union sense) does not imply the construction of the region graph.

2.6. System Under Analysis

We will generally want to refer to an implied system. In such cases we will write *SUA*, short for *System Under Analysis* to mean either the system itself or its TA model.

2.7. Reachability

Whenever TAs are used to model a system, some formalism is needed to express properties over them. As was said in Section 2.3, TCTL formulae can be used for that purpose. However, if getting the formula right is very difficult in LTL, a temporal but untimed one[3], (see, for instance, [Hol02]) the situation gets worst on TCTL. Some tools only allow for fragments of it, others resort to a combination of graphical notations, observers and reachability.

Definition 2.12 (Observer automaton)
An observer *is another TA which can be composed to the SUA and evolves to a* trap location *(also known as* error location*) whenever the behavior it monitors for takes place.*

The idea of observers for TAs was introduced in [ABL98], although its roots can be traced further away: some early articles proposed methods to generate an automaton that would have only words satisfying a certain logic formula as its language, and the use emptiness checking to decide language inclusion against the generated automaton and the one representing the SUA (e.g., [AFH91, AFH96] and others).

These trap locations on observers are labeled with a boolean property, so deciding whether the SUA has the specified behavior can be reduced to *reachability*, i.e., deciding whether there is a run in the composed automaton that reaches a state labeled with the error property.

From a tool perspective, the procedure is like this: the user specifies (typically undesired) properties in a graphical notation, which is generally easy to use. A tool automatically translates the graphical representation of the property to an observer automaton[4]. Both the observer and the SUA are fed to another tool that performs a reachability analysis.

It should be noted that the abstraction described in Section 2.4, based on symbolic states, preserves runs, so reachability algorithms can work with them.

Observation 2.2 *This thesis focuses on reachability as a behaviour validation method.*

Let's focus on reachability. We will write

- *initial state* to refer to the (symbolic) state $(\mathsf{l}_0, 0)$, where l_0 is the initial location of the SUA and 0 the zone where every clock equals zero, and
- *target state* to refer to any (symbolic) state of the SUA labeled with the error boolean property.

There are two known (and successful) algorithms to perform TA reachability [Yov97]:

- the *forward* algorithm (discussed in Chapter 5), which starts by the initial state and explores its successors, checking to see whether or not the target states have been reached, and
- the *backwards* algorithm (discussed in Chapter 4), which starts by the target states and explores its predecessors, aiming to reach the initial state.

[3]Temporal logics allow to specify precedence between events, but not specific delays, as the timed ones do.

[4]For instance, VTS [BKO05] is a graphical language (and a corresponding tool) that allows for the description of scenarios and turns them into observer automata. Chapter 9 describes it a little more.

Both can be described somehow generically in timed μ-calculus terms (see [HNSY94]). If *Init* is the set of initial states, forward is defined as

$$Reach_{fwd}(Init) = \mu S.Init \cup \bigcup_{(\mathsf{l},z)\in S} suc_{\triangleright}(\mathsf{l}, z) \text{ where}$$
$$suc_{\triangleright}(\mathsf{l}, z) = \{(\mathsf{l}', z')|\, \langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}'\rangle \in E \wedge z' = suc_{\tau}(reset_{\alpha}(z \cap \psi)) \cap I(\mathsf{l}')\}$$

The intersection between zones lifts the intersection among bounds, which can be seen as $(x_i < c_1) \cap (x_i < c_2) = x_i < \min(c1, c2)$, $reset_{\alpha}$ means putting the clocks in α to zero and $suc_{\tau}(\psi)$ means replacing the constraints of the form $x < c$ or $x \leq c$ by $x < \infty$ while leaving the rest untouched. Section 3.1 treats the topic with more precision.

The respective definition of backwards, if *Target* is the set of target states, would be

$$Reach_{bck}(Target) = \mu S.Target \cup pred_{\triangleright}(S) \text{ where}$$
$$pred_{\triangleright}(S) = pred_{\tau}(S \cup pred_{e}(S)),$$
$$pred_{e}(S) = \{(\mathsf{l}, z)|\, \langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}'\rangle \in E \wedge (\mathsf{l}', z') \in S \wedge z' = reset_{\alpha}(z \cap \psi)\} \text{ and}$$
$$pred_{\tau}(S) = \{(\mathsf{l}, z)|(\exists \delta)\ ((\mathsf{l}, z') \in S \wedge z' = z + \delta)\}.$$

Intuitively, $pred_{\tau}$ takes every state in the input set and replaces the constraints of the form $x < c$ or $x \leq c$ by $x \leq 0$ while leaving the rest untouched. $pred_{e}$, on the other hand, finds which states would have led to any state in the input set, after having traversed an edge and resetting the appropriate clocks. Again, the details are presented in Section 3.1.

Each one of the above mentioned reachability algorithms has its advantages and weakness. For instance, forward does not require the parallel composition to be built beforehand, as it can compose while exploring (this is known as *on-the-fly composition*). Backwards does not have this possibility, but it provides the basis for verification of the complete TCTL logic [HNSY94] and for controller synthesis algorithms [AT02, Pnu05].

Next chapter presents the data structures used by these algorithms.

Chapter 3

Data structures

3.1. DBMs

To represent symbolic states, a data structure called *Difference Bound Matrices* [Dil90], or *DBM* for short, is typically used. This section introduces them and is based on the presentation of [Yov97].

DBMs are $(n+1)^2$ matrices, where n is the number of clocks in the SUA. Diagonal cells are void, but all the others contain a tuple $\langle \prec, c \rangle$ called *bound*, where $\prec \in \{<, \leq\}$ and $c \in \mathbb{Z} \cup \{\infty\}$. If cell (i, j) contains $\langle \prec, c \rangle$ it means that $x_i - x_j \prec c$, where x_i and x_j are the i-th and j-th clocks in the systems (counting 0 as a special clock, used to express $x_i \prec c$ as $x_i - 0 \prec c$). The only valid use of ∞ in a cell is to mean that the difference between two clocks is unbounded. I.e., $x_i - x_j < \infty$.

The first column encodes upper bounds on clocks, while the first row encodes lower bounds. For example, if $3 \leq x_i < 7$ is encoded in a DBM z, then $z[i, 0]$ is $\langle <, 7 \rangle$ and $z[0, i]$ is $\langle \leq, -3 \rangle$ (i.e., using that $a - b > c \Leftrightarrow b - a < -c$).

There are many ways to express the same constraint system. E.g., in the following system

$$x - y < 10 \tag{3.1}$$
$$0 < x \leq 7 \tag{3.2}$$
$$0 \leq y \leq 3 \tag{3.3}$$

the first equation is not a *tight* as it can be:

$$x - y \leq 7 \tag{3.4}$$
$$0 < x \leq 7 \tag{3.5}$$
$$0 \leq y \leq 3 \tag{3.6}$$

This lack of unique representation is problematic from an algorithmic perspective because it makes it hard to decide whether two DBMs, z_1 and z_2, represent the same constraint system (i.e., deciding the thruth value of *DBM equivalence*: $z_1 \equiv z_2$). Luckily an order between DBMs can be established and a *canonical form* can be defined as the minimum of the DBMs that represent the same system.

Definition 3.1 (DBM Canonical Form)
The canonical form *of DBM z (noted $cf(z)$) is the minimum DBM that encodes the same constraints as z, according to the following order.*

$z_1 \leq z_2$ *iff for all* $0 \leq i, j \leq n, z_1[i, j] \leq z_2[i, j]$

The order on bounds is defined as follows: $<$ is strictly less than $\leq$ and $\langle \prec, c \rangle \leq \langle \prec', c' \rangle$ iff $c < c' \vee (c = c' \wedge \prec < \prec')$.

This solves the problem: checking whether $z_1 \equiv z_2$ becomes checking $cf(z_1) = cf(z_2)$. Obtaining the canonical representative, however, is quite expensive. A variation of the Floyd-Warshall All Pairs Shortest Path Algorithm [AHU89] is used, whose complexity is $O(n^3)$.

Operations on zones can be easily computed over DBMs. A detailed description on those operations can be found, for example, in [Oli94]. Here, however, we will briefly summarize the more important ones (depicted in Fig. 3.1). In what follows it is assumed that all input parameters are in canonical form.

1. Intersection: $z \cap z'$ is the zone where each cell has the stricter bound among z and z'.

$$(z \cap z')[i,j] = \min(z[i,j], z'[i,j])$$

2. Time successors: time elapsing preserves differences between clocks, because all of them advance at the same pace. Lower bounds stay the same, but upper bounds have to be pushed to infinity:

$$suc_\tau(z[i,j]) = \begin{cases} \langle <, \infty \rangle & \text{if } j = 0 \\ z[i,j] & \text{otherwise} \end{cases}$$

3. Time predecessors: lower bounds get to zero:

$$pred_\tau(z[i,j]) = \begin{cases} \langle \leq, 0 \rangle & \text{if } i = 0 \\ z[i,j] & \text{otherwise} \end{cases}$$

4. Reset successors: resetting a clock is putting it to zero, but if clock x_i is being resetted, $x_j - x_i$ should be equal to $x_j - 0$. So

$$reset_\alpha(z)[i,j] = \begin{cases} \langle \leq, 0 \rangle & \text{if } (x_i \in \alpha \wedge j = 0) \vee (x_j \in \alpha \wedge i = 0) \\ z[0,j] & \text{if } x_i \in \alpha \wedge x_j \notin \alpha \wedge j > 0 \\ z[i,0] & \text{if } x_j \in \alpha \wedge x_i \notin \alpha \wedge i > 0 \\ z[i,j] & \text{otherwise} \end{cases}$$

5. Reset predecessors: The idea is to get a zone, that, once resetted could shield z, so:

$$reset_\alpha^{-1}(z)[i,j] = \min z[k,l]| \begin{cases} x_k = 0 \wedge x_l = 0 & \text{if } x_i \in \alpha \wedge x_j \in \alpha \\ x_k = 0 \wedge x_l = x_j & \text{if } x_i \in \alpha \wedge x_j \notin \alpha \\ x_k = x_i \wedge x_l = 0 & \text{if } x_i \notin \alpha \wedge x_j \in \alpha \\ x_k = x_i \wedge x_l = x_j & \text{if } x_i \notin \alpha \wedge x_j \notin \alpha \end{cases}$$

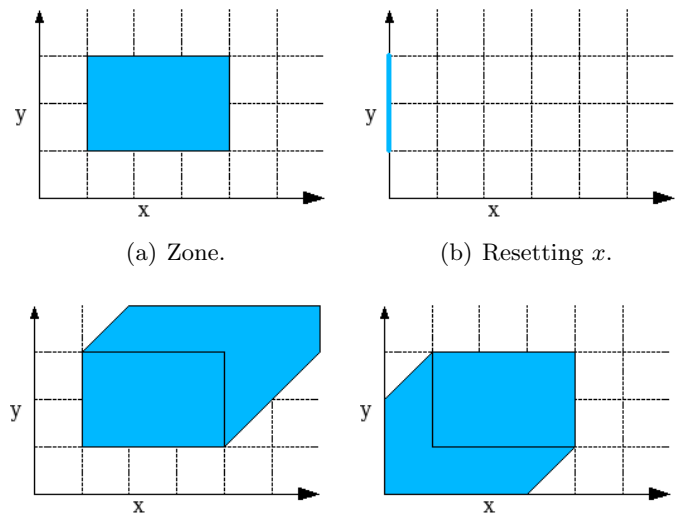
6. Union: DMBs are not closed under union, so $z \cup z'$ is {z, z'}, which is called a region[1].

Observation 3.1 *Complementing Observation 2.1, in the rest of the thesis the term* zone *will be used to refer to both the set of points specified by a constraint system as well as the data structure representing the last one.*

3.2. RDBMs

Not every constraint needs to be present for all the operations. Actually, reduced versions of the constraint system can be used most of the time, thus saving memory [LLPY03]. In practice, most tools use a variation of DBMs, called *Minimal Constraint Representation* DBMs[2] which employs that idea.

[1] Of course this is not strictly neccessary if $z \subset z'$ or $z' \subset z$.
[2] Usually called *RDBM* s, the "R" standing for "Reduced".



(a) Zone. (b) Resetting x.

(c) Time successors. (d) Time predecessors.

Figure 3.1: Different operations on the $1 \leq x \leq 4 \wedge 1 \leq y \leq 3$ zone.

As these DBMs are sparse, they are not stored like proper matrices, but as a linked list of bounds, in order to save space.

To obtain the the minimal constraint representation of a DBM, a variation of the path-recovery version of Floyd-Warshall is again used [LLPY03].

RDBMs are not new. Kronos attached a map of the minimal constraint system to each DBM and UPPAAL can be made to run using them.

3.3. Packed RDBMs

There are, however, some other ideas that can be combined. Behrmann et al. presented an abstracting technique that can be used to widen zones [BBLP04], speeding up the reachability exploration. Their technique is based on the observation that for every clock x_i there is a maximum constant against which it is compared (called c_i). So, each time a constraint of the form $x_i \prec c$, with $c > c_i$ appears on a DBM, it can be replaced by $x_i < \infty$.

This widening technique has implications for the data structure as well. If we know the maximum constant for each clock, then we know how many bits we need for each cell, and hence for each DBM.

Another widening technique comes to play. Daws and Yovine [DY96] discovered that not all the clocks are *active* in each location of the SUA, meaning that some of them can be ignored to perform operations on DBMs. Inactive clock elimination saves time, but combined with sparse storage of RDBMs, gives more sense to the idea of using a linked list representation.

Zeus combined for the first time the three ideas, thus using linked lists of cells, each one of a fixed size in bits, containing only active clocks. But having each node of the list to take a full integer would waste space. Because of that, as many of these shrunken bounds as possible are packed in each (long) integer, which is then packed in a block of continuous memory, and only then linked to another similar block.

Empirical observations show that this approach reduces time by 50% compared to a naïve linked list. This happens because less bytes are manipulated, many operations take place completely on registers and fewer pointer are dereferenced.

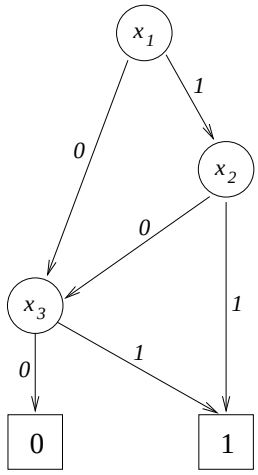


Figure 3.2: A BDD with variables x_1, x_2 and x_3.

A lot of work has been done on widening zones. The interested reader can refer to [ZLZZ03, YPD94, Bal96, BBFL03], to name a few.

3.4. CDDs

The absence of a canonical form for regions, and the inability of DBMs to express non-convex sets makes it very hard to detect superpositions, leading to repetitions of calculus and other inefficiencies. This problem was the motivation for research into new data structures. In particular, ideas like *Binary Decision Diagrams*, or *BDDs* [Bry86], seem tempting. They are successful in the untimed model checking domain (see for example, [HGGS02]).

A BDD is a directed graph $\langle V, E\rangle$ where every $v \in V$ is reachable from the root and can be a *terminal node* or not. Each non-terminal node has a index in $\{1 \ldots \mathrm{n}\}$ and two children, called *zero* and *one*. There are only two terminals, called *true* (or 1) and *false* (or 0). Non-terminals represents variables in a binary function, and their children stand for the function that is obtained once this variable is set to the corresponding value.

For instance, a BDD representing the binary function $(x_1 \wedge x_2) \vee x_3$ can be seen in Fig. 3.2.

There have been many attempts to transfer BDDs to the timed setting. See, for example, Asarin et al. [EMA+97], Strehl et al. [KL98], Møller et al. [MLAH99] among others, or Pavese [Pav06] for a thorough survey. The most successful, however, was an article by Behrmann et al. In [BLP+99] they presented *Clock Difference Diagrams* (*CDDs* for short). Their work used DBMs for most of the operations and CDDs for some of them, obtaining memory savings at the cost of extra time.

It is easy to see how CDDs leverage on BDDs: each node represents a clock, an instead of only two outgoing edges, there can be many, each of them labeled with a (pairwise disjoint) interval. In Fig. 3.3 a CDD representing a (non-convex) region can be seen.

For completeness, we mention here the work of Wang in a very similar data structure called *Clock Restriction Diagrams*, which uses open intervals (see [Wan00, Wan03a]).

In Chapter 7 we present a CDD-like data structure that was fully implemented achieving the reduction of verification times at the expense of storing some extra constraints (compared to the traditional encoding of DBMs).

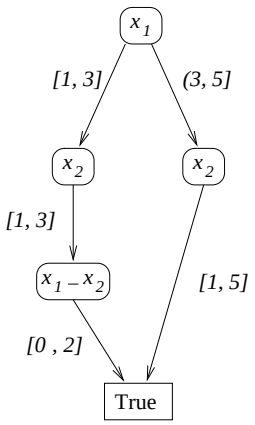


Figure 3.3: A non-convex CDD.

Chapter 4

Backwards

This chapter presents the work done on a distributed version of the *backwards* algorithm. Although this algorithm cannot perform on-the-fly composition as the *forward* one does, it provides the basis for verification of the complete TCTL logic [ACD93] and for controller synthesis algorithms [AT02, Pnu05].

Along a number of years this was our main line of work, in what regards to timed model checking. The research materialized in the first versions of ZEUS, and our results were published in [SBO02, Sch02, BOS02, BOS04a, BOS05, BOS06b].

ZEUS was developed following a software architecture centric approach. Its conceptual architecture was conceived to be sufficiently modular to house several features such as *a priori* graph partitioning, synchronous and asynchronous computation, communication piggybacking, delayed messaging and dead-time utilization.

Surprisingly enough, early experiments pinpointed the difficulties of getting speedups using asynchronous versions (Section 4.3) and showed interesting results on the synchronous counterpart (Section 4.4), although being intuitively less attractive.

Before getting into the distributed version, next section summarizes how the monoprocessor version of the backwards algorithm looks like.

4.1. Monoprocessor Version

The sequential backwards reachability algorithm, implemented in the model checker KRONOS [DOTY96], is shown in Algorithm 4.1.1. To shorten notation $s \xrightarrow{e} t$ stands for an edge with $\psi(e)$ as its guard and $\alpha(e)$ as its reset clocks. The algorithm works as follows: initially the set of target states is assigned to the R set. Then, an iterative computation is performed over the graph of (global) locations. During the computation, sets of states are symbolically represented by locations and their associated sets of DBMs. In each iteration, R is augmented by every other state that can reach it in a single step, i.e., its *predecessors*. They are computed using $pred_e$ and $pred_\tau$, the discrete and timed predecessor operators, already presented in page 13.

The process is repeated up to fixed point [HNSY94]. The computation is performed on every location and takes into account DBMs from adjacent automata nodes. The final answer is obtained checking whether an initial state belongs to R.

It should be noted that the hardest part of Algorithm 4.1.1 is the region subtraction (line 9, shown in detail in Algorithm 4.1.2). It is a highly coupled operation that requires all the zones of the *minuend* region to be operated against the complement of all zones of the *subtrahend* region. The complement of a zone is usually a non-convex set.

From a conceptual standpoint, KRONOS architecture can be viewed as a fixed point engine that

```
1: for l ∈ L | ERROR ∈ Pr(s) do
2:     R_l = I(l)
3:     ΔR_l = R_l
4: end for
5: while ∃ l' ∈ L | ΔR_l' ≠ ∅ (i.e., fixed point not reached yet.) do
6:     for l ∈ L do
7:         PredE= ⋃_{l -e-> t} pred_e(ΔR_t, ψ(e), α(e))
8:         PredT = pred_τ(PredE)
9:         ΔR_l = PredT − R_l
10:        R_l = R_l ∪ PredT
11:    end for
12: end while
```

Algorithm 4.1.1: *Backwards* reachability algorithm.

```
1: R = ∅
2: for z_1 ∈ R_1 do
3:     R_aux = copy(z_1)
4:     for z_2 ∈ R_2 do
5:         R_aux = R_aux ∩ $\overline{z_2}$
6:     end for
7:     R = R ∪ R_aux
8: end for
```

Algorithm 4.1.2: Region subtraction ($R = R_1 - R_2$).

reads and writes its non-convex sets into a regions' storage component, as depicted in Fig. 4.1.

4.2. Distributed Version

When this line of work started, a few decisions were made:

1. The tool would use Kronos libraries to perform operations on zones and regions.
2. It should run on a cluster of workstations, using message passing as communication method.
3. Automata locations would be distributed among processors (i.e., every processor knows which one handles each location).
4. As there were many unknowns regarding different options to get maximum performance, the keys ideas for the design were: software architecture based and design for change.

Item 4 was critical. To obtain a distributed timed model checker a number of issues must be dealt with. On the performance side, a delicate balance should be established: on one hand maximum distribution and parallelism is desired; on the other, minimum communication between processors should be pursued. This seems to be a very important issue, because processor cycles should be devoted to the profitable fixed point calculation, instead of house-keeping activities such as handling interruptions (context switching) to take care of message exchange. The following sections introduce some design issues and related choices. To facilitate their reading, let's start by an intuition of the workings of the distributed algorithm.

Locations are distributed among processors, which communicate by message passing. There's a distinguished one, called *coordinator*, which takes centralized decisions and is periodically informed by the others of their progress. Each processor runs a replica of the (modified) fixed point algorithm, resorting to network exchange when they need updated information that lies into the realm of some

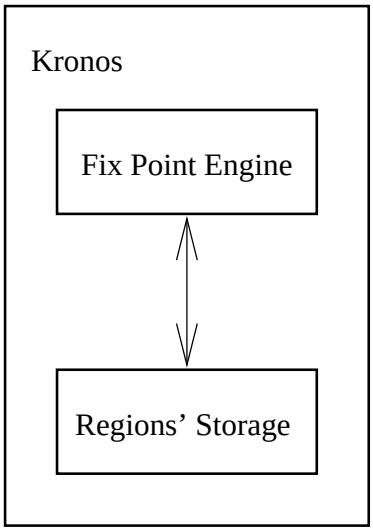


Figure 4.1: KRONOS architecture.

other processor. The above mentioned *coordinator* is in charge of detecting the fixed point, based on the periodical reports it receives.

4.2.1. Control Graph Representation

Following KRONOS approach, ZEUS also used a symbolic representation of clock values and an explicit one of the whole control graph, as opposed to the other well-known alternative: on-the-fly construction during the verification phase. This is not possible in the backwards algorithm, but also the explicit representation was an interesting starting point for experimenting different graph-partitioning strategies. It would also simplify the future implementation of dynamic location redistribution (for load balancing purposes).

4.2.2. Control Graph Partitioning

Having the right partitioning, that is, the right association between locations and processors, is vital in distributed model checking, as it determines the workload of each processing node. Proportioned workload distribution is one of the keys to speedups.

Most work in the area uses an on-the-fly control graph construction and a hashing function that tries to balance the load between processing nodes while keeping locality of calculus[1]. Having an explicit representation of the control graph opens a wide range of partitioning strategies, some of which are reported in this chapter, as listed below.

Observation 4.1 *It should be noted that although the control graph must be loaded in one machine's memory to be partitioned (i.e., the partitioning procedure is not distributed) this is generally not a problem since in the "timed world" complexity arises mainly from number of clocks and their inequalities[2], so in general, the number of locations and edges make the whole graph perfectly fit in a relatively reduced amount of memory[3].*

[1] A more in-depth comparison is presented on Chapter 6.

[2] Recall from Theorem 2.1 that the number of timed states is $O(n!2^nC^n)$, where n is number of clocks and C is the largest constant appearing in the inequalities.

[3] To have an idea of the numbers we are dealing with, current clusters have between an few an a dozen processors, with some hundred megabytes each, and the control graph is between four and twenty-something thousand locations.

We implemented the following partitioning strategies:

- Minimum Cut: We used the tool METIS [KK98] to calculate the locations' graph minimum cut. More precisely, if m is the number of edges, METIS uses an $O(m)$ heuristic approach to deal with the *NP-complete* problem of finding the minimum cut. The rationale was that the reduction in the number of edges that cross processor boundaries would minimize communication cost. Note that this partitioning does not necessarily lead to a load balance during the verification phase. In fact, this is a topological method that does not take into account the size of data structures (DBM sets) that would be associated to each location. Experiments have shown that the sizes of these sets may differ in several orders of magnitude.
- Location-cost Weighted Distribution: We weighted control locations with a metric of final workload which is directly related to the number of operations per location during an exploratory verification phase on a monoprocessor. More precisely the computational cost of a time-demanding operation, the set-theoretic difference between regions (Algorithm 4.1.2), is accumulated for each location[4]. Then, a heuristic procedure is used to evenly distribute the locations among processing nodes according to that weight. The idea is to see how a prediction on the final number region-operations performed at each location influences verification times[5]. Note that this distribution may increase communication needs since it is unaware of graph topology.
- Random Distribution: This is a uniform distribution of control locations among processing nodes. Intuitively, this kind of distribution may exacerbate communication, and looks like a bad choice. Nevertheless, it serves as the *default* distribution, considering that appropriate weight for a more thoroughly crafted strategy cannot be established without a preprocessing analysis as previously explained.

4.2.3. Level of Synchronicity

There are two major choices regarding the level of synchronicity of the computation. On the one hand, in the asynchronous version each processing node would perform fixed point calculations using the information locally available to it until reaching a (partial) local fixed point. Each time new information from other processing node arrived, the calculation would be resumed pursuing a new convergence. Fortunately, the soundness of this kind of asynchronous calculus can be proved rigorously (see Section 4.3.1). On the other hand, a synchronous version would sacrifice parallelism to mimic the calculations that would be performed by a monoprocessor version. This can be achieved by a coordinator that orchestrates global communication phases each time all processing nodes finish a fixed point iteration (that is, an iteration of fixed point calculus). It would repeat this pattern until the global fixed point was reached.

4.2.4. Communication Schema

An important decision that might affect the performance of an asynchronous version is the use of either a *push* schema, where nodes send each other regions that they will need, or a *polling* schema, where processing nodes explicitly request regions as they need them.

We implemented both strategies, with subtle variations: in the *polling* case, processing nodes basically behave as explained, but when a processor A needs a region located in another processor B, it will request all B's regions that are of interest to A –not just the one needed right now– and it will send all its regions that B needs. In principle, these mechanisms should reduce communication overhead, because fewer messages are exchanged.

[4]Please note that since regions are composed by DBMs, this cost is measured in terms of number of DBM-operations.
[5]It can be thought as an oracle-provided partition.

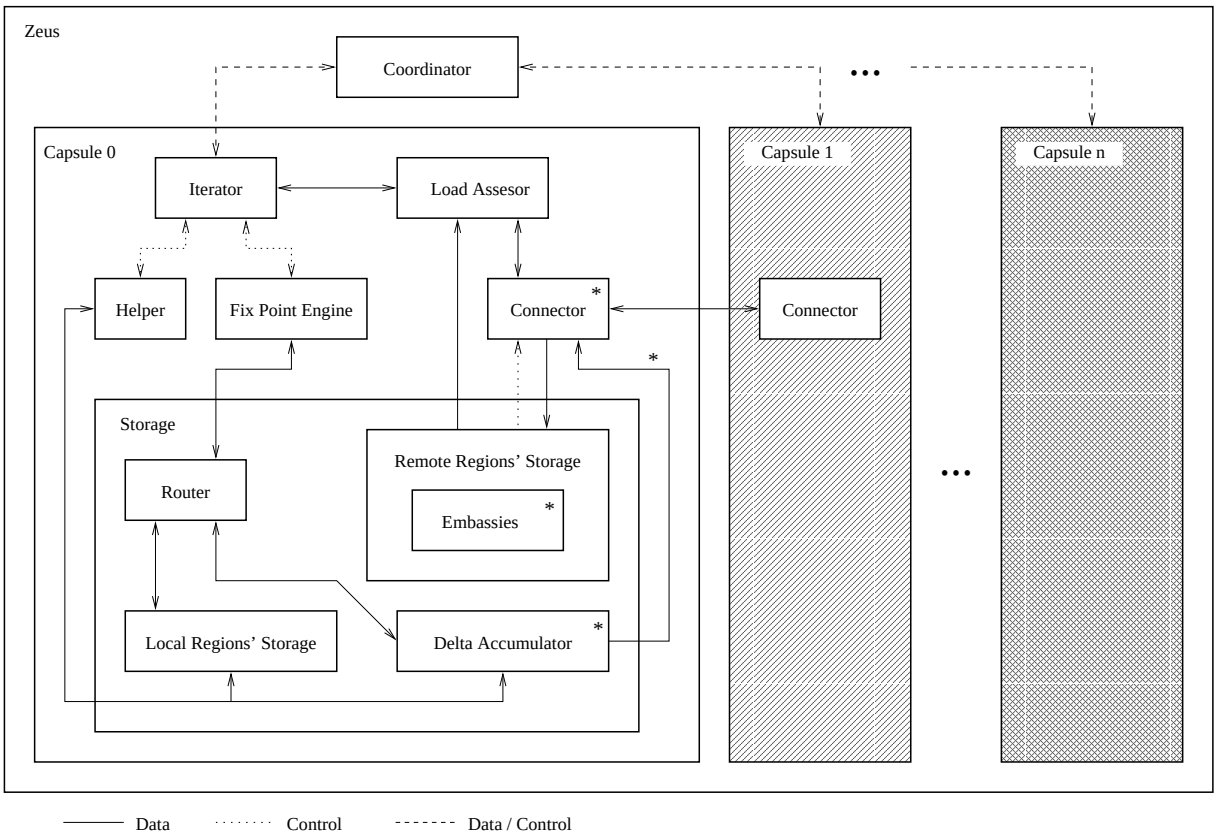


Figure 4.2: ZEUS General Conceptual Architectural View (∗ means repetition).

4.2.5. Conceptual Architectural-View

To handle all this design considerations, an architecture as shown in Fig. 4.2 was built. Its apparent complexity obeys to a key design decision: separation of concerns. It was built as a testbed for experimenting with a family of design decisions concerning synchronization, region exchange and load balance. Thus, we aimed at a loosely coupled solution where issues like fixed point calculation and the use of *polling* or *push* schema remained as independent as possible. This strategy led us to the identification of some aspects that were mapped into different components.

Each processor working in a distributed ZEUS computation is called a *capsule*. We also call *capsule* to the processes running inside the processors and their associated data structures and components.

ZEUS architecture can be more easily understood centering at the *Fixed Point Engine*. This is the component that runs the fixed point calculation, much as it does in KRONOS[6]. It reads and writes regions to the *storage* component. Each region belongs to a control location that might be assigned either to the current *capsule* or to some other. Because the *Fixed Point Engine* is unaware of such distribution, it makes no distinction between local and remote locations, except for the fact that it writes only over the local ones. Upon reception of a *read* request from the *Fixed Point Engine*, the *router* delivers regions from the *local regions' storage*, if their location is local, or from the *remote regions' storage*, if it is remote. In case of receiving a *write* request –which only happen for local regions– it stores them in the *local region's storage* and in a *delta accumulator*. There is a *delta accumulator* for each local control location that should be eventually visible from a remote *capsule*.

Again, the *Fixed Point Engine* is not aware of the existence of the *delta accumulators*; it is the *router*'s job to guarantee that written local regions also get to the accumulators whenever that information is known to be of interest of a neighbor capsule.

[6]Interested readers can find the algorithm in [BOS02].

The rationale for the existence of *delta accumulators* has to do with technical concerns regarding regions' difference. Basically, they serve the purpose of storing information instead of having to recalculate it when needed. *Delta accumulators* are emptied when regions are sent to other *capsules*.

Within each *capsule* there is an *embassy* for each neighbor *capsule*. When requested for a region, they immediately answer back if they have some yet unseen regions to provide[7].

As well as there are *delta accumulator* to benefit the sending phase of a regions' exchange, there are *embassies* to benefit the receiving phase. They are contained inside the *remote regions' storage*. One can think that there is an *embassy* at the opposite end of every *delta accumulator*. It is also responsible of showing the same content to every request during an iteration.

The memory overhead of both *delta accumulators* and *embassies* is rather small due to implementation decisions, so they scale pretty well. There is, though, some repetition of content, specially in *embassies*, that increases with the amount of connected *capsules*. The minimum cut partitioning approach keeps this number low, so this is not a problem in that case, although, potentially, it could arise in the others. It should also be noted that experimental observations placed the per-processor memory overhead in an acceptable range (see Fig. 4.7).

A *capsule* A has a $connector_{A,B}$ iff there is at least one edge between a location in A and a location in B. *Connectors* handle network communication. That is, given two *capsules*, A and B, there might be many edges in the control graph going from locations assigned to A to locations assigned to B, call it k. Although A can have up to k *delta accumulators* and k *embassies*, it has only one *connector* handling bi-directional communication to B. The same happens at B.

The *helper* performs data representation compression (i.e., representing regions with lesser DBMs if possible). Compression is applied in the local repository and the *delta accumulator*. This leads to a "semantically innocuous" dead time utilization that hopefully will make future computations and messaging lighter.

The *coordinator* starts the process, partitions the graph, distributes the workload and establishes whether global fixed point has been reached or not. It also collects statistics. It receives information from the *iterators* on the *capsules* to make decisions.

A formal description of the architecture including state machines and transducers can be found in [Sch02].

4.3. Asynchronous Version

The *push* schema was easily implemented by sending information stored at the *delta accumulators* as soon as possible.

A key point for our *polling* schema was the determination of the *regions-demand mode* for each *capsule*. More precisely, the *load assessor* component was in charge of determining whether the *capsule* was idle or not and whether its regions-demand was high or low. It decided in which of the following three possible modes the computation was currently in:

Low regions-demand, busy processing node In this state the processor is busy with the fixed point calculation and can go along without requesting newer regions from another processing node, thus leaving the chance that a future request will bring a larger region, due to the monotonicity of the calculus.

High regions-demand, idle processing node In this state the processing node has reached a temporary fixed point, so it is idle and its requests are immediately delivered.

[7]For technical reasons special care is taken to exhibit the same information to every request belonging to the same *Fixed Point Engine* iteration.

High regions-demand, busy processing node This state serves the purpose of predicting the previous one. The processing node is not yet idle, but an heuristic[8] predicts that it will be soon, so its requests are sent without delay.

When an *embassy* needs new regions from a remote *capsule*, it contacts the appropriate *connector*. In a naïve *polling* approach, a *connector* merely proxies requests from one *capsule* to another. However, a few twists were made for performance, i.e., for minimizing interruptions to the fixed point calculation at the other end. First, when requested to send a petition, the *connectors* check with the *load assessor* to see if there is a "high regions-demand", in which case they proceed requesting regions to the target *capsule*. However, if there is a "low regions-demand", they will silently drop the request. Secondly, *connectors* do *piggybacking*: instead of asking for a particular locations' region, they ask for the regions of the full set of locations they represent; also, to "compensate" the other *capsule* for the interruption, they send the regions of the full set of locations that are of interest to the *capsule* receiving the request. These regions come from the corresponding *delta accumulators*. The receiving *connectors* immediately answer requests, regardless of their own *capsule* regions-demand mode. When the new regions arrive to the demanding capsule they become visible to the *Fixed Point Engine* at the next iteration.

As was noted before, a *capsule* can be idle, meaning that global fixed point is not yet reached because there is still work going on at other *capsules*, but a local fixed point was established anyway. To take advantage of the idle time, a new component, the *helper*, is introduced. It performs auxiliary tasks, such as representation compaction of the regions at the *delta accumulators* and at the *local regions' storage*.

Finally, an *iterator* is responsible for deciding to run either the *Fixed Point Engine* or the *helper* based on the readings of the *load assessor* and the arrival of new information. It is also responsible for sending statistics to the *coordinator*.

Hence, the *coordinator* is the component that detects that fixed point or that initial states have been reached.

4.3.1. Correctness Proof

To prove correctness we follow a common strategy for reasoning about distributed algorithms by splitting the proof in two parts: (a) the semi algorithm described converges to the minimum fixed point, and (b) the *coordinator* will eventually detect fixed point when it actually happens.

To address the first issue we resort to the concept of *Asynchronous Iterations*, due to Cousot [Cou78].

Definition 4.1 (Asynchronous Iterations)

Let $(P, \sqsubseteq)$ be a complete lattice, n a positive integer and $F : P^n \to P^n$ a monotonic operator.

Let Ord be the set of ordinal numbers and $\langle c^\delta : \delta \in Ord\rangle$ be a sequence of elements of $\mathbb{N}_n = \{1, \ldots, n\}$ such that:

a) $(\forall \delta \in Ord)(\forall i \in \mathbb{N}_n)((\exists \alpha \geq \delta)(i = c^\alpha))$

Let $\langle \tau^\delta : \delta \in Ord\rangle$ be a sequence of elements of Ord^n such that:

b) $(\forall i \in \mathbb{N}_n)(\forall \delta \in Ord)(\tau_i^\delta < \delta)$

[8] The heuristic is the determination that the number of regions changed per iteration is decreasing below certain threshold.

c) $(\forall \delta \in Ord)(\forall i \in \mathbb{N}_n)(\exists \beta \geq \delta)((\forall \alpha \geq \beta)(\delta \leq \tau_i^\alpha)))$
d) $(\forall \beta, \delta \in Ord)((\beta \text{ is a limit ordinal } \wedge \beta < \delta) \implies ((\forall i \in \mathbb{N}_n)(\beta \leq \tau_i^\delta)))$

An Asynchronous Iteration *of the operator F and the sequences $\langle c^\delta : \delta \in Ord\rangle$ and $\langle \tau^\delta : \delta \in Ord\rangle$, starting from $D \in P^n$, is defined as follows:*

$$\begin{array}{lcll} x^0 & = & D & \\ x_i^\delta & = & x_i^{\delta-1} & \forall \text{ successor ordinal } \delta \wedge \forall i \neq c^\delta \\ x_i^\delta & = & F_i(x_1^{\tau_1^\delta}, \ldots, x_n^{\tau_n^\delta}) & \forall \text{ successor ordinal } \delta \wedge \forall i = c^\delta \\ x_i^\delta & = & \bigsqcup_{\delta' \leq \delta} x_i^{\delta'} & \forall \text{ limit ordinal } \delta \end{array}$$

In [Cou78] it is proven that Asynchronous Iterations are stationary sequences and their limit is indeed the least fixed point of operator F starting from D.

That is, Asynchronous Iterations provide a model of iterative computing that guarantees that if hypotheses (4.1.a)-(4.1.d) are met, then the minimum fixed point will be reached even if the computation does not follow the standard order of fixed point iterative calculus. In particular, during the computation described by this model, data from different past instants might be used. This is the kind of phenomenon that arises in distributed environment, where remote data used in a given calculus may have changed at the original site. More precisely, x can be understood as a shared memory (distributed or not) of n positions featuring minimal mutual exclusion mechanisms. Then, being $\langle \delta \rangle$ a growing sequence on logical instant, $\langle c^\delta \rangle$ will be a sequence stating that the cell x_i (where $i = c^\delta$) is being written by some processor. The value to be written by the processor is calculated as the application of the functional F to $x_1^{\tau_1^\delta}, \ldots, x_n^{\tau_n^\delta}$ where each τ_k^δ stands for the last time when the processor read x_k.

Let's go back to the hypotheses needed to prove convergence. Firstly, (4.1.a) requires fairness of computation (i.e., each cell is visited infinitely often). Condition (4.1.b) simply requires calculations to be based on past instants. Roughly speaking, (4.1.c) and (4.1.d) require that the information should eventually (in a finite number of steps) be ready for utilization in future calculations.

Now let's see how runs of our algorithm can be characterized as Asynchronous Iterations and match these hypotheses.

Let ζ be an arbitrary run of our algorithm where at each logical step at most a cell (a control graph location in our case) is written[9]. Then, we define $\langle c_\zeta^\delta : \delta \in Ord\rangle$ as the sequence containing the cells written at each logical instant for the run ζ. On the other hand, $\langle \tau_\zeta^\delta : \delta \in Ord\rangle$ is defined as follows:

$$(\tau_\zeta)_i^\delta = \begin{cases} \text{the last instant when the } i\text{-th cell} & \text{if } \delta \in \omega \\ \text{was read previous to } \delta \text{ step in } \zeta & \\ \delta - 1 & \text{if } \delta \notin \omega \end{cases}$$

The definition for values beyond ω is arbitrary in order to make the proofs easier. Indeed, fixed point point calculus is done over a finite lattice and it is not necessary to iterate beyond omega. Thus, being $pred_\triangleright$ the basic monotonic operator over the lattice used for backwards calculus, what follows is a characterization of runs of the algorithm:

$$\begin{array}{lcll} (x_\zeta)^0 & = & \bot & \\ (x_\zeta)_i^\delta & = & (x_\zeta)_i^{\delta-1} & \forall \text{ successor ordinal } \delta \wedge \forall i \neq c_\zeta^\delta \\ (x_\zeta)_i^\delta & = & pred_{\triangleright i}((x_\zeta)_1^{(\tau_\zeta)_1^\delta}, \ldots, (x_\zeta)_n^{(\tau_\zeta)_n^\delta}) & \forall \text{ successor ordinal } \delta \wedge \forall i = c_\zeta^\delta \\ (x_\zeta)_i^\delta & = & \bigsqcup_{\delta' \leq \delta} (x_\zeta)_i^{\delta'} & \forall \text{ limit ordinal } \delta \end{array}$$

[9]This linearization is possible with no loose of generality.

Actually, the real run of our algorithm is a proper subsequence of x_ζ since the number of steps does not go beyond ω. Now let's show that x_ζ is indeed an Asynchronous Iteration (and thus it will converge to $\mathit{fixedpoint}(\mathit{pred}_\triangleright)(\bot)$ as expected). Let's analyze each condition of Definition 4.1.

a) Trivially holds. Each location is assigned to a *capsule* and *capsules* are executed continually in a fair way wrt. its locations.
b) Trivially holds. When $\delta \in \omega$, in order to calculate a new application of the functional, each *capsule* resorts to data stored in its repository (local or remote is irrelevant) and the information contained there was generated in a past instant. When $\delta \notin \omega$, by definition, $(\tau_\zeta)_i^\delta$ is $\delta - 1$.
c) Let x_i (without loss of generality) be a location assigned to *capsule c*. We would like to show that given a δ there exists a future instant β after which every *capsule* that will require x_i will use a value written at or after instant δ. Therefore we need to show that sooner or later (the instant β) *capsules* are able to "see" changes that have happened after δ. Using simple fairness hypothesis, and assuming that the net is not lossy, in the push schema this is clearly bound to happen. For the polling schema, the argument is subtler. There are two simple cases: (a) the information produced at logical-time δ arrives in a piggybacked message, or (b) the reading capsule enters a high regions-demand mode, the request will eventually be made, and thus updated info will return as answer. The only remaining point to analyze is the fact that sooner or later either the information arrives piggybacked or it is claimed by *capsules* that need to read regions of the given location. More precisely, the only scenario that can jeopardize the property is an infinite arrival of new information from neighbor *capsules* that disable a given *capsule* from entering a high regions-demand mode (and thus keeps it from claiming the information produced after instant δ). However, this is not feasible due to the fact that each arrival implies a new piece of information of the lattice which is indeed finite (due to Theorem 2.1).
d) Trivially holds. If β is a limit ordinal then $(\forall i \in \mathbb{N})((\tau_\zeta)_i^\delta = \delta - 1)$ (by definition of τ_ζ). On the other hand, if $\beta < \delta$ then $\beta \leq \delta - 1$.

Now we have proved that x_ζ converges and it is not difficult to see that it converges in less than ω steps like the original algorithm since there is a finite number of values that x_ζ^δ can hold and $\mathit{pred}_\triangleright$ is a monotonous operator.

The last question to solve is: "Does the *coordinator* correctly detects the fixed point?". On one hand, premature abortion is not really possible since not only local fixed point is required to stop the calculus, but also the *delta accumulators* must be empty and the number of sent messages must be the same that the received ones, network-wise. On the other hand, since the previous predicate is periodically checked, after convergence the global fixed point will eventually be detected by the *coordinator*.

4.3.2. Experimental Results

The experiments of this section were run on a cluster consisting of 9 Linux 2.4 workstations connected through a 100 Mbps Ethernet network, each one running on a 1.6 GHz Pentium IV processor, with 128 MB of RAM. Case studies are described in Appendix A.

For *RCS5*, the monoprocessor version took 110 and 230 s to verify the reachable and unreachable cases, respectively. The asynchronous version behaved erratically in the case studies, even showing important slowdowns with respect to the monoprocessor version. For instance, a 2-processor run of an unreachable case, using a subscription policy and METIS as partitioning strategy took 554 seconds to complete (more than twice as much the monoprocessor times). In the rest of the examples the same behavior was observed. E.g., the *CB7,2* example only finished with 4 processors being 10 times slower than the monoprocessor version.

In principle, these results seemed counterintuitive since asynchronous strategies were meant to

make good use of parallel processing resources. However, there was an explanation for this unsatisfactory behavior that is deeply rooted in the nature of the verification problem. First, in the asynchronous versions, the sequence of sets of regions computed to reach the fixed point point may differ to the sequences built by the monoprocessor version. Secondly, region representation as sets of zones is not canonical[10]. Then, the order in which operations are applied can affect the cardinality of the set that represents the resulting region – that is, the number of DBMs involved. The experiments exhibited an important boost in the number of DBMs required to represent intermediate regions during the asynchronous fixed point calculus and consequently an increase in the cost of operations between regions. This phenomenon is called *fragmentation* and some authors reported a similar problem when breadth-first traversal is not honored to calculate fixed points [Beh05, BLO02].

Going back to the *RCS5* case-study, in a monoprocessor version the number of operations obtained is approximatedly 697 millions while the calculus of the 2-processor asynchronous version produces approx. 1.3 billions. It is also worth mentioning that I/O had little impact in total verification times, and partition and distribution strategies affect fragmentation in an unpredictable way.

These observations motivated the implementation of the synchronous policy, described in next section. Due to Zeus' modular architecture, the effort was very limited, restricted to the *iterator* and *coordinator* components.

4.4. Synchronous Version

Experiments reported in the previous section (and in more detail in [BOS02]) revealed some counterintuitive bad behavior of the asynchronous version with respect to the number of operations and total time required by Kronos. The problem being data-structure fragmentation caused by a non-standard evaluation order, we decided to emulate the way Kronos calculates the fixed point. This motivated the design and implementation of a synchronous policy.

Although it lost potential parallelism, it mimicked the operations that Kronos performs on a monoprocessor. Therefore, if load balance was achieved, this strategy may have provided important speedups wrt. the monoprocessor version. The basic idea behind the synchronous version was quite simple: the *coordinator* gave the pace at which all the other *capsules* work. Verification methods like the one presented in [GM02] use a similar approach.

The *coordinator* waited for all the *capsules* to have finished one iteration of their fixed point calculation and then authorized them to go into a *region-exchange phase*. Once they were done, they resumed the computation again.

The *iterator* component in each *capsule* implemented the other half of the protocol (see Algorithm 4.4.1). The tricky part was knowing that it was done exchanging regions. This was accomplished by evaluating the following predicate: "All my sending buffers are empty and I have received at least one message from each adjacent *connector*". Please note that each *capsule* went into the iteration phase once it was done with the region exchange, which was a local decision (the *coordinator* does not needed to authorize this). This allowed for the fixed point calculation to start as soon as possible, taking advantage that not every *capsule* had dependencies on every other *capsule*.

4.4.1. Correctness Proof

The correctness of the synchronous version leverages on the asynchronous one (Section 4.3.1), but the proof is easier since every fixed point iteration is followed by an exchange phase. Moreover, in a synchronous version a stronger property holds: at the end of each iteration the resulting set of regions is the same one that would be obtained by the monoprocessor version.

[10] Recall from Definition 3.1 that there may not be a unique way to represent the same set of clock values.

```
while ¬GlobalEnd do
    change = fixedpoint_engine.iterate()
    notify_phase_end(Iteration, change)
    wait_for_coord_clearance(Exchange)
    exchange_regions()
    notify_phase_end(Exchange)
end while
```

Algorithm 4.4.1: ZEUS synchronous iteration.

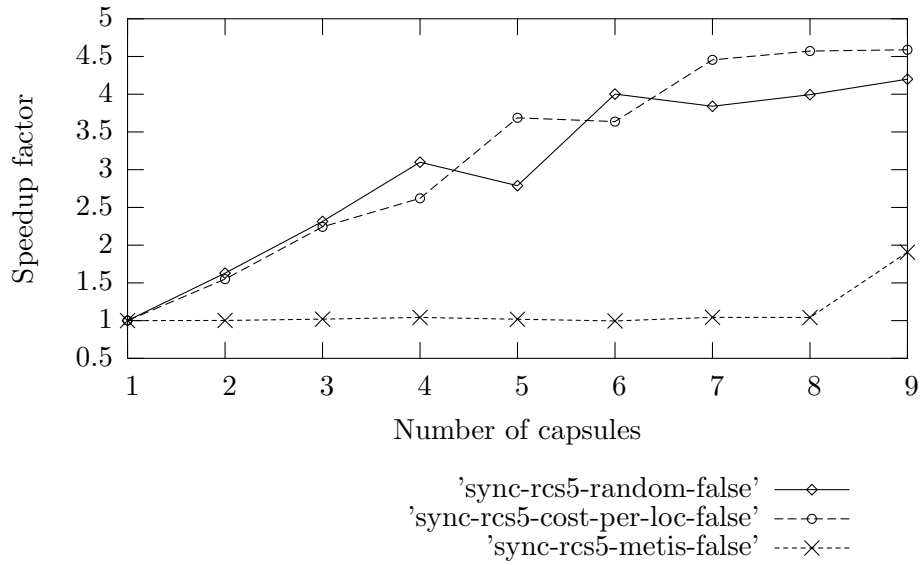


Figure 4.3: Speedup of the Synchronous Version for the unreachable case of *RCS5*.

4.4.2. Experimental Results

Results obtained were much more satisfactory, evidencing predictable speedups (see Fig. 4.3 and Fig. 4.4).

To explain why a linear speedup is hard to achieve, we defined a metric of the processing time wasted in each experiment as follows. Being $\#cap$ the number of *capsules* and $\#it$ the number of iterations, we get:

$$
\begin{aligned}
WastedTime &= \sum_{i=1}^{\#it} \sum_{c=1}^{\#cap} Waiting_{i,c} \\
\%waste &= \frac{WastedTime}{(TotalTime - I/O\ time) \times \#cap} \times 100
\end{aligned}
$$

This measured the time potentially wasted due to unbalanced workload. In Tables 4.1 and 4.2 we can see the verification times and corresponding $\%waste$ metric for the different partition strategies.

For instance, the poor performance of the Minimum Cut version can be explained by the unbalanced workload produced by not taking into account that there are huge differences in the number of DBMs associated to each location. Indeed, in this example less than 2 percent of locations are responsible for more than 90 percent of effort in the fixed point calculation. Random distribution does not perform significantly worse –sometimes even performing better– than location-cost weighted distribution. This fact seems awkward at first sight, but can be accounted for by carefully analyzing the iteration-level statistics.

Table 4.1: %*waste* for example *RCS5* (unreachable "Error" state).

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	230.68	0.00	230.68	0.00	230.68	0.00
2	141.56	17.37	143.88	19.08	228.27	49.93
3	99.68	20.79	99.44	20.45	224.41	66.88
4	74.43	19.50	85.09	31.45	219.46	74.86
5	82.73	44.35	60.48	21.03	224.73	80.90
6	57.64	32.05	61.31	37.84	229.71	84.65
7	60.06	46.53	50.05	36.14	219.28	86.83
8	57.75	53.22	48.77	43.28	219.61	89.04
9	54.92	61.17	48.59	52.94	119.94	82.84

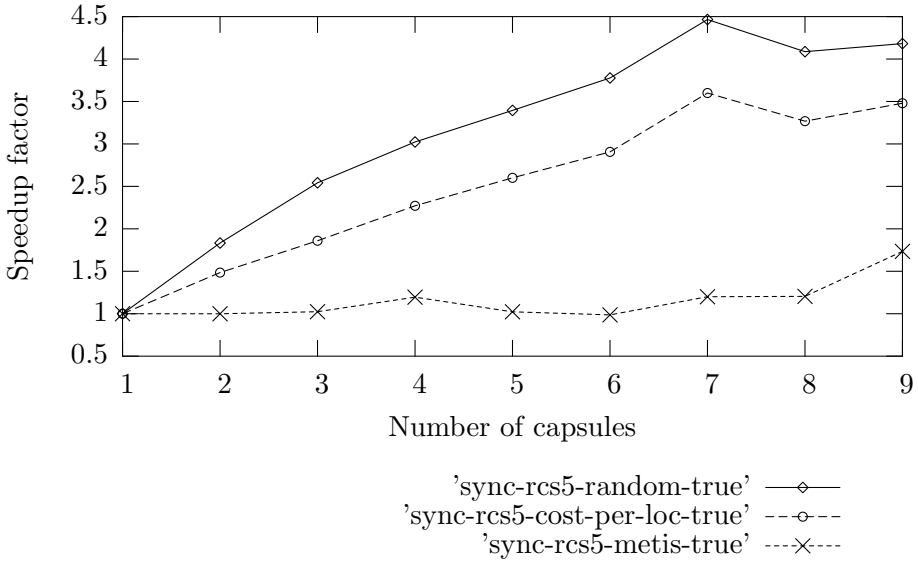


Figure 4.4: Speedup of the Synchronous Version for the reachable case of example *RCS5*.

Table 4.2: %*waste* for example *RCS5* (reachable "Error" state).

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	110.40	0.00	110.40	0.00	110.40	0.00
2	60.22	0.97	74.12	33.95	109.98	49.90
3	43.44	3.00	59.16	39.07	107.34	66.53
4	36.52	8.24	48.44	33.80	91.93	69.99
5	32.52	14.59	42.30	36.14	107.44	79.84
6	29.23	16.43	37.85	33.20	111.44	83.09
7	24.72	8.76	30.56	22.66	91.56	82.49
8	27.01	31.74	33.66	44.47	91.26	84.90
9	26.40	39.86	31.61	51.64	63.37	79.42

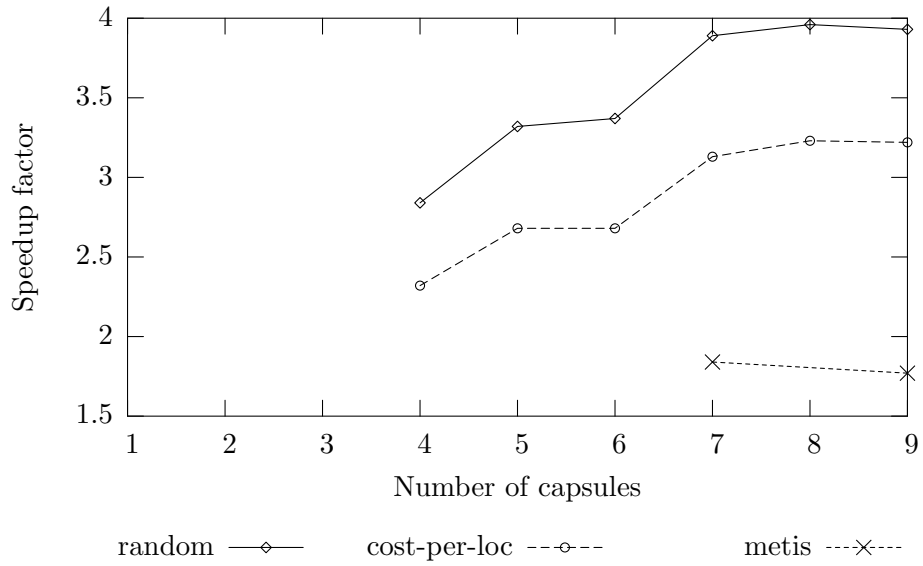


Figure 4.5: Speedup of the Synchronous Version for the unreachable case of example *CB7,2*.

Table 4.3: %*waste* for example *CB7,2* with an unreachable "Error" state.

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	Ran Out of Memory	0.00	ROM	0.00	ROM	0.00
2	ROM	N/A[11]	ROM	N/A	ROM	N/A
3	ROM	N/A	ROM	N/A	ROM	N/A
4	71.97	17.10	88.14	40.88	ROM	N/A
5	61.71	16.29	76.20	43.60	ROM	N/A
6	60.76	23.27	76.38	50.06	ROM	N/A
7	52.59	19.79	65.46	47.50	111.14	74.93
8	51.71	29.22	63.38	48.30	ROM	N/A
9	52.02	35.76	63.57	54.89	115.52	81.74

Partitioning based on the location-cost metric was founded on the conjecture that the cost reported by the metric has the *same distribution* among the iterations for every location. We empirically observed that equally distributing the final cost among the capsules does not imply equally distributing it at every iteration. In a worst-case scenario, this could lead to a "sequentialization", meaning that at each iteration only one capsule works, but the work-load still seems balanced in the long run.

A promising path seemed to be the construction of a *dynamic* partition strategy that could deal with this kind of phenomenon.

Similar results were obtained with *CB7,2*: while one machines' memory was insufficient, multi-processor runs showed some speedup, and behaved as expected with regards to the different distributions, as can be seen in Tables 4.3 and 4.4, and Figures 4.5 and 4.6. We performed a run in another machine apart from the cluster with 512 MB. In this one it took 203.27 and 204.59 s to finish in the reachable and unreachable cases respectively, and was used as the exploratory run of Location-cost Weighted distribution.

Example *CB7,2* was selected to illustrate the memory usage pattern. Fig. 4.7 shows its memory footprint during the last iteration for the minimum number of processors required to verify the model (four) and the maximum number of available processors (nine). As can be seen, the whole system

[11] For technical reasons we are not able to get data from partial runs, i.e., runs that abort due to lack of memory.

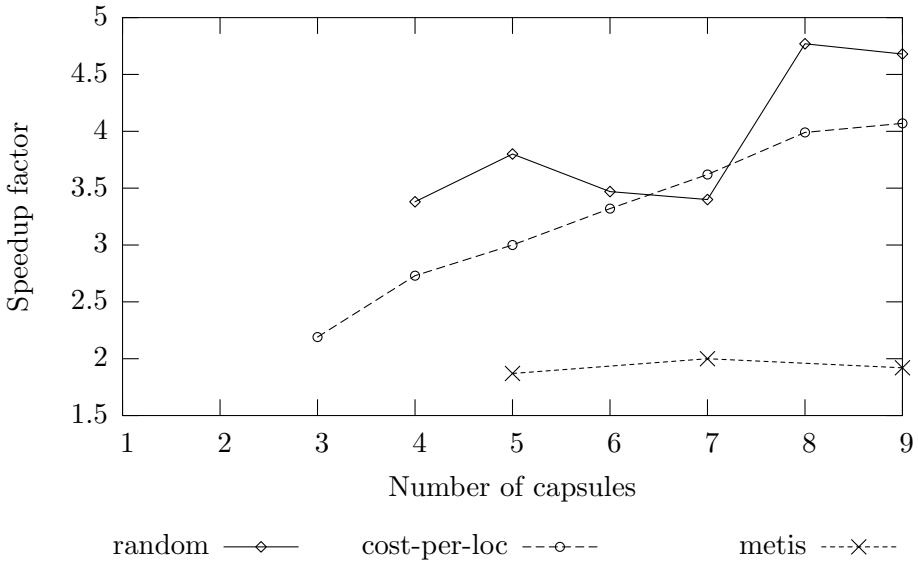


Figure 4.6: Speedup of the Synchronous Version for the reachable case of example *CB7,2*.

Table 4.4: %*waste* for example *CB7,2* with a reachable "Error" state.

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	ROM	0.00	ROM	0.00	ROM	0.00
2	ROM	N/A	ROM	N/A	ROM	N/A
3	ROM	N/A	10.22	39.93	ROM	N/A
4	60.07	10.22	16.12	42.18	ROM	N/A
5	53.43	16.12	43.34	46.79	108.74	72.17
6	58.56	43.34	61.27	47.94	ROM	N/A
7	59.86	53.36	56.09	51.66	101.43	77.92
8	42.58	29.27	50.93	54.95	ROM	N/A
9	43.39	38.03	49.92	50.78	106.08	84.23

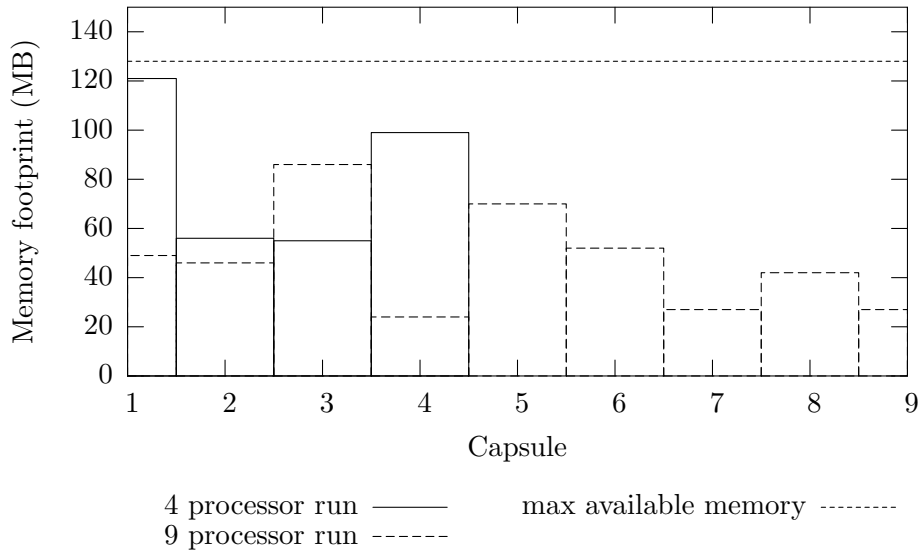


Figure 4.7: Memory footprint of the *CB7,2* example.

Table 4.5: %*waste* for example *FDDI7* with an unreachable "Error" state.

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	ROM	0.00	ROM	0.00	ROM	0.00
2	ROM	N/A	ROM	N/A	ROM	N/A
3	71.11	1.93	97.91	39.52	81.68	32.59
4	58.15	5.49	73.14	36.69	ROM	N/A
5	63.56	12.00	76.41	51.62	86.47	59.22
6	46.41	4.47	56.71	42.33	82.47	65.24
7	38.86	3.23	51.03	43.96	65.46	69.83
8	37.97	4.83	49.37	47.16	80.12	73.32
9	34.16	4.91	45.02	49.95	63.57	74.55

uses more memory than available on each single machine. The memory usage is uneven, but this is not completely unexpected, since the %*waste* metric already shows uneven workload distribution.

The four-processor run required 331 MB while the nine-processor required 423 MB, leading to a per-processor overhead of only 18 MB, that is, roughly a 5.5%. As stated in Section 4.2.5, this is a reasonable price payed for having local copies of some data required for the calculus.

A similar course of action was taken for the *FDDI7* example. We also performed an exploratory run outside the cluster, obtaining 182.36 s for the unreachable case and 93.43 s for the reachable one. The results are presented in Fig. 4.8 and Fig. 4.9, and Tables 4.5 and 4.6.

This case shows sharper speedups because of its symmetric nature. It's interesting to see how some runs exhibit low or no waste of time at all (meaning that the workload distribution was quite fair) and even though the speedup factor is not linear (see for example 9 processor, random distribution in both Fig. 4.8 and Fig. 4.9). These examples showed what the actual overhead of distribution (in terms of I/O, region interchange, etc.) could be in some cases, particularly when iterations were relatively fast.

Nevertheless, no definitive conclusions could be drawn about it until a strategy such as dynamic distribution was implemented to deal with the most pressing obstacle: balancing the load among the processors in the general case. Next section shows the work done on a dynamic backwards algorithm.

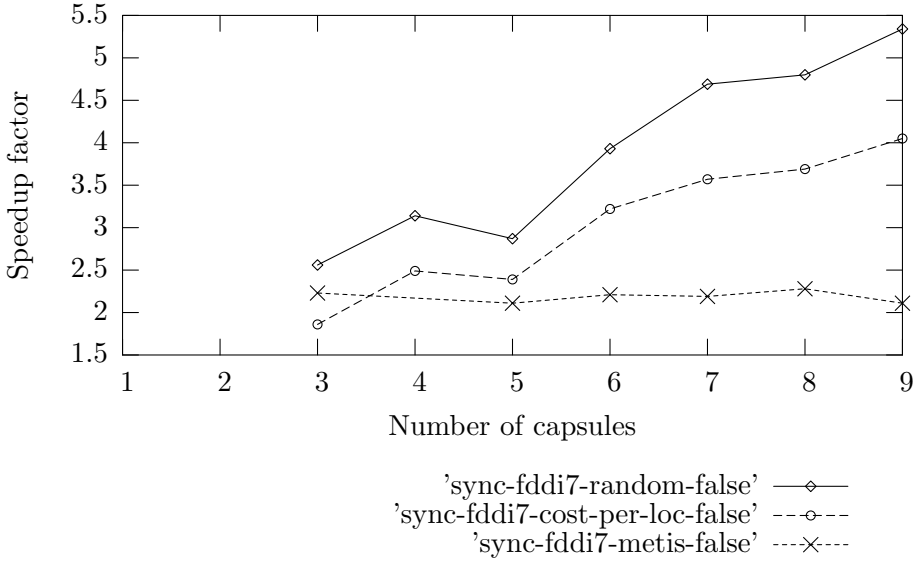


Figure 4.8: Speedup of the Synchronous Version for the unreachable case of example *FDDI7*.

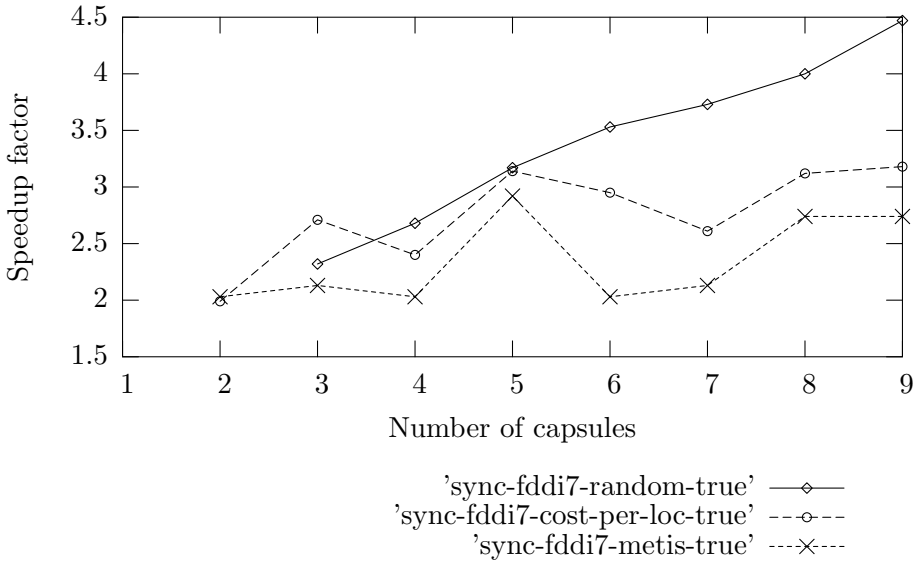


Figure 4.9: Speedup of the Synchronous Version for the reachable case of example *FDDI7*.

Table 4.6: %*waste* for example *FDDI7* with a reachable "Error" state.

# capsules	Random		Location-cost Weighted		Min Cut	
	total time	%*waste*	total time	%*waste*	total time	%*waste*
1	ROM	0.00	ROM	0.00	ROM	0.00
2	ROM	N/A	46.99	0.00	46.05	1.41
3	40.21	5.19	34.51	48.68	43.79	31.88
4	34.81	2.12	38.97	37.29	45.99	48.59
5	29.43	2.90	29.73	57.39	32.01	57.39
6	26.44	0.00	31.64	45.74	46.09	63.01
7	25.08	1.12	35.80	54.56	43.87	68.32
8	23.36	0.00	29.97	66.46	34.06	71.73
9	20.90	0.00	29.36	51.85	34.11	69.24

Table 4.7: Number of locations taking at least 1% of the workload.

Example	# *HL* / total	$\Sigma_{l \in HL}$ workload(l)
MinePump	5/4452	$56\% + 16\% + 8\% + \ldots = 92\%$
RCS6	32/5288	$2\% + 2\% \ldots + 1\% = 62\%$
Conveyor4AB	8/11240	$21\% + 20\% + 20\% + 20\% + \ldots = 89\%$
Conveyor6A	2/1344	$60\% + 28\% = 88\%$
RS-BR	13/22710	$12\% + 9\% + 3\% + \ldots = 44\%$
RS-C	4/25975	$63\% + 2\% + 1\% + 1\% = 67\%$

4.5. Dynamic Version

4.5.1. Intrinsic Problems

Very unbalanced iterations, specially the latter ones where the number of involved zones is larger, could mean that in practice only a few processors are doing most of the work while the rest are idle, undermining the time previously gained and producing a worse result in the overall process. To dig into the reason for this behavior, we focused on the nature and evolution of the workload pattern.

Table 4.7 presents this information for the "heaviest" iteration in each example, where *HL* stands for the set of *heavy locations*. We considered a location *heavy for an iteration* when it required at least 1% of the workload of that iteration. The value 1% was arbitrary but seemed a good threshold to explain a Pareto-like behavior (the sum of heavy locations workload was, in most of our examples, greater than 60% of the total workload as shown in the tables).

Table 4.8 shows some more details about the workload patterns of Table 4.7. For each case study, we picked the three most consuming iterations as representatives of the behavior. Iteration number (k) is on the second column and the contribution of the iteration, measured as the percentage of the iteration workload over the total, on the third. The fourth column represents the percentage of the workload for iteration k that corresponds to locations in HL_k, the set of heavy locations for iteration k. Next column contains the workload in iteration k attributable to locations that were defined as heavy in the previous iteration. The last column shows how many locations varied at least 50% in their percentual contribution to the workload over the ones that belong to both HL_k and HL_{k-1}, that is, these are the ones that were significant in two consecutive iterations but nevertheless varied substantially from one iteration to the other.

Observation 4.2 *Some challenging observations could be drawn from Tables 4.7 and 4.8:*

1. *Firstly, during each iteration the processing effort was monopolized by the manipulation of a relatively small set of heavy locations, which turned out to be the ones with larger data structures in terms of number of zones. This conditioning seems intrinsic to the models, and, as discussed later, establishes a practical limit on scalability for these sort of approaches that do not split and distribute data structures of locations.*
2. *Secondly, the set of heavy locations changes from one iteration to the other. Sometimes that change was relatively smooth (e.g.,* MinePump, Conveyor4AB, Conveyor6A*) but there were cases where the workload distribution radically varied from one iteration to the other (e.g.,* RS-C, RS-BR*). Even when the constitution of the heavy set from one iteration to the next had little or no change, the individual workload contribution of locations may vary substantially (e.g.,* MinePump, Conveyor6A*).*
3. *Finally, heavy locations tend to be associated to some observer locations (i.e., distribution of heavy locations by observer projection is not uniform). In general, the highest numbered location correspond to the target states and the ones with the lowest number are further from it. Usually*

Table 4.8: Details of heavier iterations.

Example	k	Contrib(k)	%workload(HL_k, k) (# HL_k)	%workload(HL_{k-1}, k) (# HL_{k-1})	#(varied significantly) / #($HL_k \cap HL_{k-1}$)
MinePump	25	25.35%	96.59% (08)	87.28% (11)	4/4
	22	17.66%	97.31% (03)	94.23% (13)	1/2
	24	15.48%	94.53% (11)	76.24% (12)	5/6
RCS6	7	41.05%	78.16% (32)	78.16% (32)	0/32
	6	39.22%	61.40% (32)	61.40% (32)	0/32
	5	10.63%	41.63% (32)	0% (00)	0/0
Conveyor4AB	9	20.21%	88.24% (04)	92.01% (12)	0/4
	11	16.01%	88.46% (08)	88.46% (08)	4/8
	12	15.91%	91.22% (11)	86.38% (08)	0/7
Conveyor6A	17	39.66%	96.50% (07)	93.33% (08)	1/5
	16	19.23%	92.42% (08)	85.86% (06)	2/4
	18	15.23%	98.43% (05)	98.43% (07)	1/5
RS-BR	9	67.57%	45.41% (13)	18.24% (21)	1/3
	8	15.03%	35.22% (21)	17.28% (16)	1/5
	7	11.07%	49.28% (16)	35.74% (17)	3/7
RS-C	17	62.99%	79.97% (04)	0.26% (03)	0/0
	16	26.04%	72.76% (03)	2.13% (11)	0/0
	15	5.09%	25.49% (11)	12.31% (10)	1/2

the lowest numbered locations of the observer take the most work. Fig. 4.10 shows workload associated to each of the observer locations in the MinePump *example. Despite the number of reachable SUA locations per observer node, the workload is quite uneven and monopolized by some observer locations. The other examples exhibited a similar pattern. This phenomenon will be revisited in Section 4.5.4.*

■

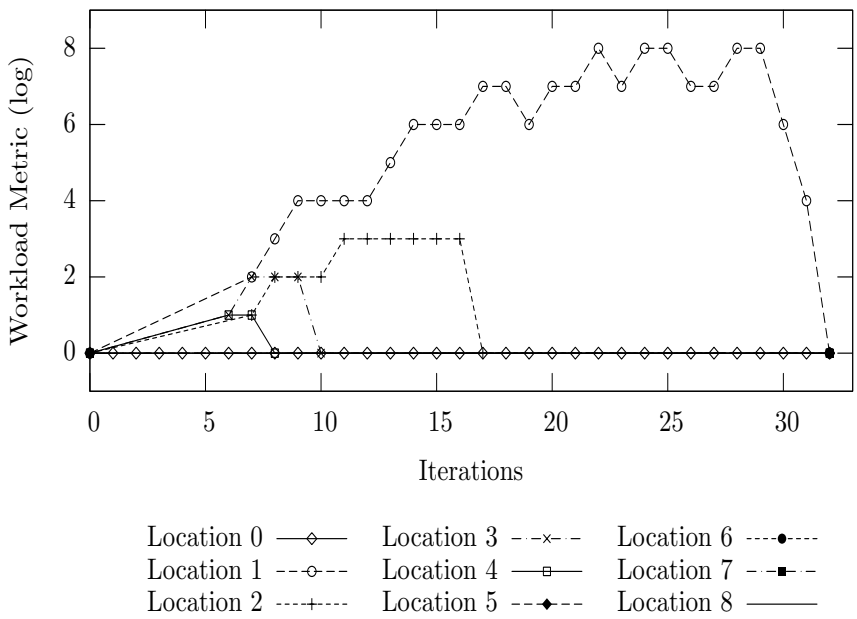


Figure 4.10: Workload per observer location on *MinePump*.

The second observation meant that the concept of fair distribution needed to be established relative to each iteration. Thus, this justified the use of mechanisms for migrating locations (and their future computational work) from one processor to another in each iteration, trying to evenly distribute the workload during the whole verification.

Unfortunately, efficiently distribute and manipulate zones associated to locations into several processing nodes (as the first observation suggests) seems hard to achieve in the backwards verification settings. Indeed, region differentiation used in line 9 of Algorithm 4.1.1 (and detailed in Algorithm 4.1.2) requires, in principle, performing a DBM operation between each pair of zones in the Cartesian product between R_s and the complement of $PredT$.

We thought that it was therefore worth trying to push forward the limit of the presented approach, at least in some common verification scenarios. That is, we aimed at improving workload distribution and processor utilization as much as possible without splitting regions. Next section relies on the third item of Observation 4.2 and a migration strategy based on profiling previous verification sessions on similar models. Together, they improved the performance of the approach reaching reasonable results in the light of the findings presented here.

4.5.2. Redistribution, Workload-Profile Reuse and Observer-Induced Partitioning

In order to make the most out of the current approach, we based our analysis on two important observations about how model checking is usually applied:

- To reflect improved (faster) components, longer delays or stricter timing requirements it is usual

to make small changes to some clock comparisons in the model.
- In several verification scenarios, models are actually built from the composition of a SUA with an observer, as already mentioned in Section 2.7.

The importance of these two assumption will become apparent in Section 4.5.4, after some specific background is presented.

4.5.3. Dynamic Zeus

This section summarizes the results obtaining while trying to use a prediction strategy.

As already shown in Table 4.8, there were important variations of the workload between iterations, even in the subset of the most heavy locations. These variations jeopardized the balance, thus incrementing %*waste* and consequently the verification times, in the general case. This is why the ability to migrate locations among processors was built into Zeus, in what we called the *dynamic* version.

Dynamic Location Redistribution

Moving locations on-the-fly poses several challenges. This subsection focuses on the technical aspects of the process, postponing the consideration of what, when and where to migrate.

Migration decisions were made by the *coordinator* at the end of each iteration and broadcasted to the *capsules* along with the clearance to exchange regions (line 6 of Algorithm 4.4.1).

Capsules received a mapping of locations to their new *capsule* (at line 4 of Algorithm 4.4.1) and reconfigured their interconnections accordingly. For each location migrated, the originating *capsule* needed to examine its successors, in order to detect which *connectors* should be kept active (even adding new ones). This was important because having received updates from every *active connector* was the trigger to declare the end of the exchange phase (line 6 of Algorithm 4.4.1). To make things harder, some *connectors* had to be flagged temporary active, meaning that they had to be kept alive just to migrate locations, but needed to be shutdown immediately afterwards.

The *connectors* neededed to be reconfigured, so they knew which locations they were actually serving. Because the topology and boundaries changed, there could be *connectors* to some other *capsules* no longer needed, and new ones might had to be created. In the last two cases, besides reconfiguring *connectors*, new *embassies* might have needed to be instantiated, or old ones destroyed.

More importantly, in many cases region's data for the affected location needed to be exported by a *capsule* and imported by another. The corresponding locations were marked, and said regions were exchanged during the region exchange phase (line 5 of Algorithm 4.4.1). Once finished, the migrations list was processed again, so clean up activities could take place, including the removal of former local regions.

The result of the process had to be that each *capsule* was reconfigured as if the current partition had never changed since the beginning. Although conceptually simple, implementation details made it quite involved. All this reconfiguration time had to be added to the transmission delay, implying that migrations were not free, and should be minimized whenever possible.

Coordinator's Decision

Two problems had to be solved to make good migration policies: the workload for the next iteration had to be predicted, and based on that prediction a fair repartitioning had to be found. An important concern is minimizing migration time, because a "perfect" balance made no sense if it required a significant delay due to network transmission and, predominantly, the operations described in the previous paragraphs.

This version used the ParMETIS library [SKK00] which is based on heuristic methods to handle efficiently graph repartitioning problems. ParMETIS tries not only to balance weight while minimizing number of movements, but also to reach minimum cut. Although this would seem like a good idea because it minimizes communication, it is not such a pressing issue in a synchronous environment over a fast local area network. It should be pointed out that a number of good methods exist for rebalancing in the untimed scenario (see [HGGS02, NC97] among others). Unfortunately they are not directly applicable as they usually don't have to deal with locations having different (unsplittable) weights.

However, prediction for the next iteration was still unaccounted for.

Workload Prediction

In [BOS04b] we described an approach based on on-the-fly prediction of the cost of the heaviest operation performed by the *Fixed Point Engine*: region subtraction (line 9 of Algorithm 4.1.1). The reader is referred to the aforementioned article for details as only an outline of the method will be given here. The subtraction operation has two parameters ($PredT$ and R_s), and its computational complexity depends on its sizes. Although the second is known for the next iteration, the size of $PredT$ is not. In order to obtain reasonable values for it, the following mechanism was developed, based on the intuition that many calculi are usually repeated and some that are not have only slightly different inputs: each time the real operation was performed, the size of its parameters was rounded to its most significant digit, and was stored along with the size of the result. For example, if two regions of size 23456 and 337 were subtracted, obtaining a region of size 128, the tuple $\langle 20000, 300, 128\rangle$ would have been stored. If a new operation has the same rounded parameters size, the old values are overwritten, on the assumption that recent values would predict better future results.

When the real value needed to be estimated, its rounded estimated parameters sizes were looked up in the collected information. If no match was found, a default value was used. It should be noted that this is kind of a simplistic approach to best-fit matching, and more sophisticated algorithms could have been used, but it had the advantage of having a small overhead in terms of both space and time, and featured good results on its own.

Some case studies showed interesting speedups when using the prediction method. These included *RCS6* and *MinePump*. There were others, however, where accurate predictions could not be made. In the cases where the number of heavy locations was very small and responsible for an important share of the total workload, a failure to predict those locations' associated work immediately translates into a completely wrong migration set (cft. Section 4.5.1).

In Section 4.5.4 a different approach is presented.

4.5.4. Working Around the Limitations

Section 4.5.1 presents the problems we believe to be central to the distribution of the backwards model checking algorithm for timed automata, provided that regions are not splitted between processors. The next two sections present some observations that complemented the technology presented in Section 4.5.3 to work around them, in the sense that improved results could be achieved in specific settings.

Observer-Based Partitioning

An interesting observation could be made over models where the property to verify is expressed via an observer: their topology shares certain characteristics which we detail below.

As was said, in several verification scenarios, models are actually built from the composition of

a SUA with an observer (recall Section 2.7). Observers can be built in a number of ways: by hand, automatically from formulae [ABL98] or automatically from graphical patterns using tools like VTS (presented in [ABKO04], but see also Chapter 9). In particular VTS-generated observers are almost a DAG (except for self loops). Roughly speaking, the shape of the observer is like this: an initial location where most of the SUA actions take place, then a few locations meant to synchronize with the negative scenario to be matched, and then the error location. This shape is imposed on the composed system.

As this version of Zeus reachability calculus was backwards, the first global locations with some non-empty regions would be the ones corresponding to the last observer locations (typically *ERROR* labeled). Some iterations later, previous observer locations (i.e., those "closer" to the initial one) would start to accumulate non-empty regions; but because of the many different transitions guards and invariants that had "filtered" those target regions, they would be more fragmented. That in turn meant that the first locations would be heavier to compute in each successive iteration. As was advanced in Section 4.5.1, experiments seemed to confirm the supposition (recall Fig. 4.10).

Taking this phenomenon into account, as well as the previously mentioned non-trivial cost of migrations, evenly distributing the most heavy observer locations among the *capsules* seems to make sense. Fig. 4.11 shows the speedup of various case studies, for both the traditional round-robin partitioning as well as the new observer-induced distribution. The experiments were run on 9 machines with a 300 MHz R12000 processor, running Irix with 1 GB RAM each.

As can be seen in Fig. 4.11, some examples (e.g., *RCS6* and *RS-BR*) showed an erratic behavior on the round-robin distribution, but performed very well on the observer-induced. Others, while not close to the desired lineal speedup, were still better than their round-robin counterpart in the overall performance.

Taking Advantage of Common Usage Patterns

This subsections aims to answer the question of whether a previous run can be used as oracle for load-balancing decisions if only some parameters of the model are changed. It experiments with the idea of migratig workload according to the aforementioned previous-run oracle.

In order to reflect improved (faster) components, longer delays, stricter timing requirements or some similar variations, it is common to make small changes to some clock comparisons in the model, as was advanced at the beginning of Section 4.5.2. Although these modifications can drastically change the output of the verification, in usual cases found in practice, they have little impact. In those verification scenarios, information from previous runs can be used to create an initial fair partitioning, and to migrate locations based on (believed) trustable workload data.

The procedure was as follows: migrations were produced offline based on the workload data of the unmodified system –called *original* from now on– and using ParMETIS to calculate them. It was instructed to avoid migrations if the workload was below a configurable threshold. That typically meant that the first iterations were migration free. To improve things a little bit, the initial partition was created as the previously mentioned observer-induced one, modified by the first non-empty migration.

An interesting point to note is that this fast offline preparation could also compute a theoretical %*waste* metric and compare it to the %*waste* metric of the original run, on an iteration basis. Two purposes were served by this comparison: first, if the theoretical waste was more that the actual run, the migrations could be discarded for that iteration. More importantly, if during the run an important divergence between the theoretical and the real per-capsule metric was detected, the tool could conclude that the change in the model was indeed important, making the model (very) different from the original, and thus disregarded the remaining migrations.

In order to exercise these ideas all the models were modified as follows:

- *RCS6*: the controller was made two time units slower.

- *Conveyor4AB* and *Conveyor6A*: the task that starts when an object is detected and ends when it is lifted from the conveyor belt was given a stricter deadline by ten time units.
- *RS-BR* and *RS-C*: the WCCTs of the readers were relaxed and unequaled.
- *MinePump*: the WCCT of the watchdog was strengthened.

Analyzing preliminary performance of the modified models with observer-induced partitions vs. observer-induced partitions plus migrations based on the workload data of the original example, we noticed that naïve migrations did not seem to add value, even decreasing performance on some cases. For instance, *Conveyor4AB* required 3216 seconds on a monoprocessor and 1627 in two processors without migrations. If naïve migrations were used, the two-processor verification required 3859 seconds, i.e., more than the original. We found the explanation for that phenomenon again in Table 4.7. Case studies where the number of heavy locations was small would suffer an important migration overhead. This was because ParMETIS needed to move many of locations to compensate for one of these, and as was said earlier, each migrated location required non-trivial amounts of work, independently of its region's size, thus increasing the total verification time.

To counter this downside, we refined the migration strategy: let ParMETIS do its job, but then only migrate (as instructed by ParMETIS) the most heavy locations. This would of course produce a more unbalanced workload pattern, but the expected gain was to compensate that with a smaller migration overhead. Another heuristic idea was used: as ZEUS checks for the presence of the initial (timed) state in the reached set in each iteration (that is, not only when the fixed point is reached), if the expected outcome was "true", then it does not make sense to move locations on the (expected) last iteration, because that would only delay the moment when the initial location was processed.

Table 4.9 shows the times for the case studies with migrations, versus only an observer-induced partition. For each case study the original and modified versions are shown. Round-robin (RR) and observer-induced (OI) partitioning are included for both versions, and also observer-induced plus migrations (OI+migr) for the modified one.

In some cases –notably *RS-BR* and *RS-C*– a decrease in time is not followed by a decrease in %*waste*. This is caused by the difference in I/O times involved, consequence of different communication requirements due to changed partition boundaries (cft. %*waste* formula in Section 4.5.1). Fig. 4.11 shows the speedup gained: the migration method improved performance on most of the cases.

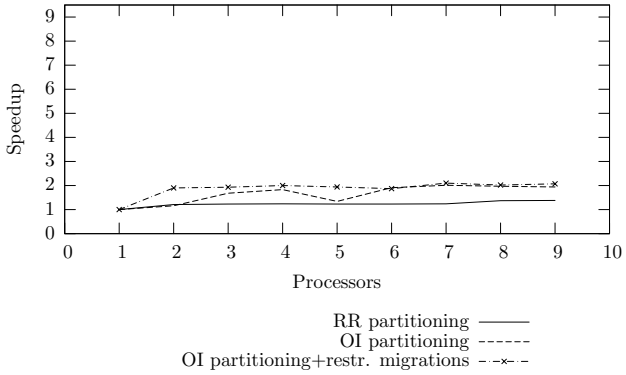
Example	Distrib.	Processors								
		1	2	3	4	5	6	7	8	9
MinePump	RR	13.17	10.63	10.43	10.54	10.56	10.59	10.46	9.45	9.50
			(41)	(60)	(70)	(77)	(80)	(83)	(84)	(85)
Original	OI		11.29	7.83	7.16	9.78	6.87	6.53	6.66	6.73
			(44)	(47)	(57)	(74)	(70)	(73)	(77)	(80)
MinePump	RR	13.25	11.54	7.88	7.45	10.12	7.23	7.12	7.08	6.91
			(45)	(47)	(59)	(80)	(75)	(75)	(78)	(82)
Modified	OI		11.44	7.87	7.23	9.92	6.94	6.60	6.72	6.82
			(44)	(46)	(57)	(74)	(70)	(73)	(77)	(80)
	OI+migr		6.98	6.85	6.61	6.82	7.08	6.31	6.56	6.42
			(30)	(42)	(55)	(65)	(70)	(74)	(78)	(79)
RCS6	RR	6.91	3.64	2.43	2.41	3.70	3.13	2.44	1.75	1.29
			(07)	(08)	(31)	(64)	(65)	(61)	(53)	(43)
Original	OI		3.67	2.51	1.87	1.66	1.30	1.44	1.12	1.13
			(08)	(11)	(10)	(19)	(13)	(33)	(25)	(34)
RCS6	RR	5.41	2.85	1.89	1.90	2.86	2.45	1.90	1.38	1.01
			(07)	(08)	(31)	(64)	(65)	(61)	(53)	(43)
Modified	OI		2.85	1.95	1.45	1.27	1.00	1.09	0.86	0.86
			(08)	(11)	(10)	(18)	(13)	(31)	(24)	(32)
	OI+migr		2.75	1.81	1.37	1.21	1.02	0.90	0.76	0.75
			(04)	(04)	(05)	(12)	(15)	(13)	(11)	(21)

Example	Distrib.	Processors								
		1	2	3	4	5	6	7	8	9
Conveyor4AB	RR	3.23	2.31	1.55	1.54	2.32	2.28	1.55	1.51	0.81
			(31)	(31)	(48)	(73)	(77)	(71)	(74)	(56)
Original	OI		1.63	1.56	1.49	1.45	1.42	0.82	0.78	0.84
			(02)	(31)	(45)	(54)	(61)	(43)	(48)	(56)
Conveyor4AB	RR	3.21	2.35	1.56	1.55	2.33	2.30	1.56	1.53	0.82
			(31)	(31)	(48)	(73)	(77)	(70)	(74)	(56)
Modified	OI		1.63	1.56	1.48	1.44	1.42	0.82	0.77	0.84
			(02)	(31)	(44)	(55)	(61)	(43)	(47)	(57)
	OI+migr		1.52	1.28	0.92	0.76	0.72	0.73	0.71	0.71
			(02)	(28)	(10)	(14)	(24)	(36)	(43)	(49)
Conveyor6A	RR	5.30	3.43	3.37	3.34	3.37	3.38	3.28	3.28	3.26
			(26)	(49)	(62)	(70)	(75)	(78)	(81)	(83)
Original	OI		3.61	3.38	3.50	3.34	3.26	3.32	3.26	3.11
			(29)	(50)	(64)	(70)	(74)	(79)	(81)	(82)
Conveyor6A	RR	5.36	3.45	3.34	3.36	3.44	3.42	3.32	3.31	3.30
			(26)	(49)	(62)	(70)	(75)	(79)	(81)	(83)
Modified	OI		3.62	3.38	3.52	3.34	3.26	3.32	3.26	3.11
			(29)	(50)	(64)	(70)	(74)	(79)	(81)	(82)
	OI+migr		2.36	3.24	3.16	3.18	3.27	3.05	2.96	3.05
			(28)	(49)	(61)	(68)	(74)	(78)	(79)	(82)
RS-BR	RR	9.71	4.16	3.10	2.55	2.65	2.58	2.04	1.95	1.58
			(19)	(14)	(34)	(68)	(66)	(63)	(60)	(53)
Original	OI		4.68	2.91	2.49	1.89	1.55	1.43	1.24	1.20
			(10)	(17)	(27)	(40)	(32)	(36)	(37)	(44)
RS-BR	RR	9.59	4.13	3.09	2.54	2.63	2.55	2.02	1.92	1.55
			(20)	(16)	(35)	(68)	(66)	(63)	(61)	(54)
Modified	OI		4.63	2.87	2.46	1.85	1.52	1.40	1.24	1.20
			(10)	(17)	(27)	(40)	(32)	(37)	(38)	(44)
	OI+migr		4.61	2.95	3.49	2.42	2.10	1.73	1.10	1.02
			(10)	(11)	(29)	(27)	(20)	(29)	(39)	(36)
RS-C	RR	4.76	1.62	1.38	1.32	1.33	1.27	1.19	1.13	1.06
			(40)	(53)	(64)	(75)	(75)	(78)	(80)	(82)
Original	OI		3.91	1.44	1.39	1.32	1.17	1.16	1.10	1.07
			(39)	(54)	(64)	(71)	(75)	(78)	(80)	(82)
RS-C	RR	4.77	1.64	1.40	1.31	1.35	1.30	1.25	1.20	1.07
			(40)	(53)	(63)	(75)	(75)	(78)	(80)	(82)
Modified	OI		3.93	1.46	1.38	1.33	1.19	1.17	1.09	1.08
			(40)	(54)	(65)	(70)	(75)	(78)	(80)	(82)
	OI+migr		3.91	1.68	1.49	1.15	1.11	1.06	1.03	1.02
			(39)	(51)	(63)	(71)	(75)	(77)	(80)	(82)

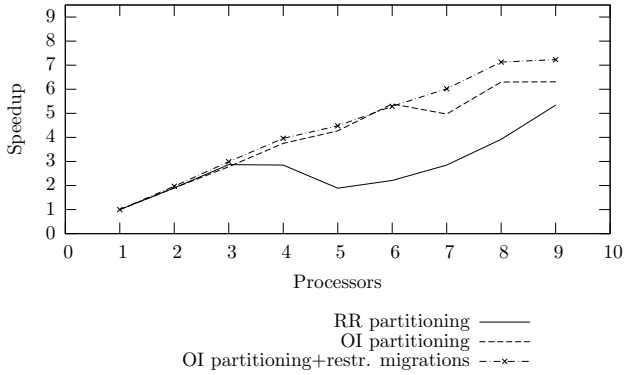
Table 4.9: Total time in kilo seconds and (%*waste*) obtained for different distributions.

Observation 4.3 *Based on the information summarized in Table 4.7, we postulate that these speedups are almost as good as they can get without splitting the workload per location. This has to do with the fact that it is not reasonable to expect good performance on n processors if there are only m locations monopolizing most of the workload (with $m < n$).*

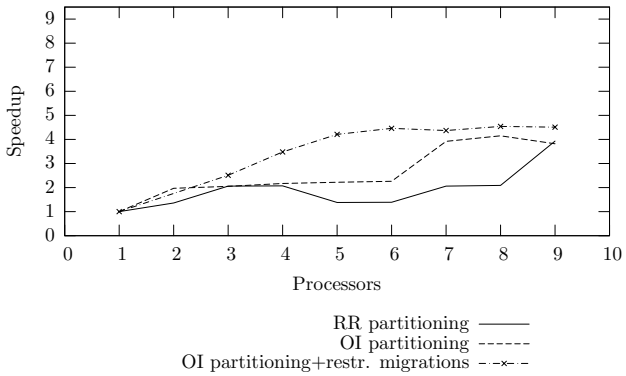
More precisely, Table 4.7 justifies why the quasi-linear speedup cannot go beyond a certain threshold. For example, *Conveyor4AB* has four locations taking approximately 20% of the work each one



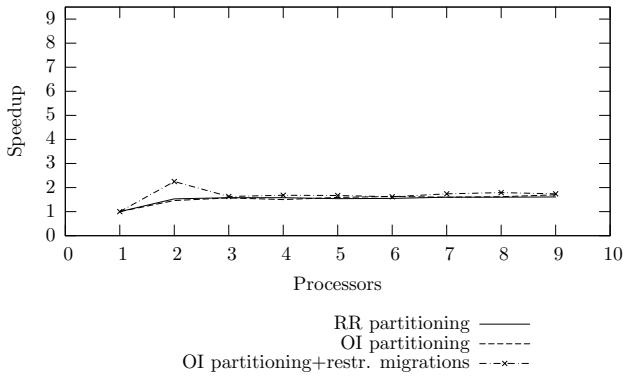
(a) *MinePump*.



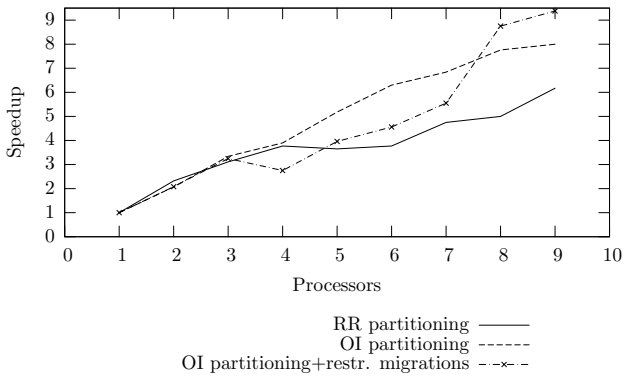
(b) *RCS6*.



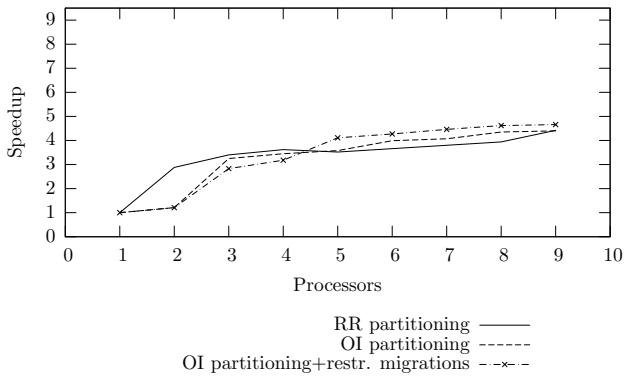
(c) *Conveyor4AB*.



(d) *Conveyor6A*.



(e) *RS-BR*.



(f) *RS-C*.

Figure 4.11: Speedups for observer-induced (OI) partitions with and without migrations for the constraint-modified case studies.

on its heaviest iteration. Considering that in many cases (such as this one) the heaviest iteration conditions the total time, this means that there is no current way to achieve a speedup substantially greater than five, in this example.

However, if in the future ZEUS were run on a mixed network of workstations, some of them single processor, some of them multi-, it would make sense to use migrations to take heavy –nowadays unsplittable– locations to a multiprocessor where various kernel threads can work on it at the same time.

It should be pointed out that the combined method is far superior than the naïve round-robin distribution reported in [BOS05].

There is another interesting advantage to be gained in these kind of scenarios where a verification is run over a model slightly different from a previous one: from the data of Table 4.7 an estimation can be made on the number of processors to use. That is, intuition might suggest to use as many processors as available, but if the model is really similar to the previous one, it might make sense to use only a subset of them, freeing the remaining ones for some other use.

The aforementioned idea will become specially important once the next generation of multi-core processors becomes popular. It is common for engineers to make incremental changes to the model and leave the model checker in background while they continue working on other tasks. With this information at hand, the model checker can be prevented from consuming all the processing power of the workstation without degrading its own performance.

4.6. Conclusions

Though asynchronous computation of fixed point is theoretically sound, DBM-based data structures may be inefficient when the exploration order differs from that of the monoprocessor version. This justifies synchronizing iterations in each processing node, thus mimicking the sequential algorithm. Unfortunately, this also leads to wasted times because of processing nodes waiting for the signal that authorizes the next iteration, which is sent just after all processing nodes have finished the current fixed point iteration. Thus, load balance becomes the key factor to achieve speedups in practice.

However, as Section 4.5.1 shows, some locations monopolize the total workload, meaning that close-to-linear speedups are not feasible without splitting locations among processors, which in turn seems impractical for the backwards calculus in a distributed environment (i.e., no shared-memory).

This point signal a quasi-*end of the road* for the distribution of the backwards algorithm.

Nevertheless, the backwards version of ZEUS seems to get the most out of the availability of processors given this essential limitation of the location-based approach. Preliminary observations lead us to the supposition that this parameter seems to be related to the number of feasible simple paths in the analyzed design. For instance, the more parallel activity (represented by interleaving actions) the better suited for distribution using ZEUS-like techniques and the more scalable its verification. This seems an interesting property since it suggests that the location-based approach may be useful for highly asynchronous designs. This is an idea that might be worth exploring in the future.

Two workarounds are presented. Firstly, we leverage on the fact that observer-based analysis of a design is a common practice in verification, and accordingly distribute locations evenly wrt. the accompanying observer location. Secondly, when the verification engineer analyzes a new version of the system where only deadlines and delays are modified, we explore how to reuse effort metrics collected for the original model. This technique achieves reuse by migrating locations among processing nodes trying to balance the likely effort in each iteration, without incurring in a high redistribution overhead.

On one hand, experiments show that these heuristics have some positive impact on load balance

and some interesting speedups are achieved using small sized clusters. On the other hand, as previously mentioned, the number of locations associated to most of the computation effort during each iteration (m) still sets a practical limit on the expectation to get linear speedups for a given quantity of processors (i.e., n processors would be of no help if m is less than n), without splitting regions. When the reuse technique is applied, that number may be useful to suggest to the verification engineer the number of processors that should be involved for an efficient model checking of a system variant. We consider this finding, and specially the possibility of estimating it a priori on the reuse scenario, a key one.

This version of Zeus was distributed, but with the next generation of multi-core processors becoming popular and our finding that only a limited number of locations are responsible for most of the workload (recall Table 4.7), it might make sense to develop a parallel-distributed hybrid version. It would work in a distributed cluster, but multi-processor nodes would be multi-threaded, parallelizing the work also on the location level. In that settings, migrations might play a key role, because heavy locations could be moved to dense-processor nodes in each iteration. Work is underway on such direction, and preliminary results are promising.

Future work agenda also includes, on one hand, revisiting the asynchronous fixed point computation strategy while changing the underlying data structure to avoid the before-mentioned fragmentation phenomena (some initial work in this sense is reported in Chapter 7).

Looking back, a lot was learned. Chapter 5, featuring the forward algorithm leverages much of the insight gained here.

Chapter 5

Forward Model Checking

This chapter discusses the approach known as *forward reachability*, where verification starts from the initial states and tries to reach the target ones. The first section introduces the basic algorithm. After that, Section 5.2 presents the basic distributed version of the algorithm. In this kind of verification workload balance is also extremely important, so in Section 5.3 an approach for migrating workload on-the-fly is introduced, and its correctness is shown.

Data from the implementation of the above mentioned strategy is shown in Section 5.4, just before our conclusions and future research plans on the topic.

5.1. Forward Reachability

The basic (conceptual) procedure for checking reachability of property ϕ is straight forward. It requires a queue of states to be explored, commonly known as *Pending*, and a set of already visited states, known as *Visited*. The algorithm works like this: insert the initial state in the *Pending* queue and initialize an empty *Visited* set. Then, while *Pending* is not empty, take a state from it and check whether the property ϕ holds for it. If it does, finish with a "YES" result, otherwise, put it in *Visited*, compute its (timed) successors (one for each outgoing transition). The total number of states is finite, as guaranteed by Theorem 2.1, but there can be repetitions. To ensure termination, before putting them in *Pending*, it should be checked that they are not *included* in any other state from *Visited*. In an untimed exploration the check would be simpler: is the same (explicit) state already *present* in *Visited*? In the timed framework, clocks values conform multi-dimensional polyhedra. As such, if a new state is included in an already explored one there is no point in revisiting it, even if it is not exactly equal. The pseudocode is shown in Algorithm 5.1.1, where the *Pending* set is revisited while not empty on line 6, and the successors of the current state are queue in the for-loop of line 8.

We will usually refer to the contents of *Pending* as *to-be-explored states*.

In the innermost cycle resides the IsStateAlreadyVisited?$((\mathsf{l}, z))$ function, which checks to see if there is a state (l, z') in *Visited* such that $z \subseteq z'$. Actually, a performance conscious implementation requires some more work: if $z' \subset z$ then (l, z'), although already visited, can be taken out the *Pending*. This is why, following [BDLY03] the two queues contain pointers to the same set of elements. The details of this algorithm are shown in Algorithm 5.1.2. The check to see if some state can be taken out of *Pending* is featured in line 7.

An important characteristic of the forward algorithm is that it composes on-the-fly. That is, its input are the TA components, and the composition takes place in line 8 of Algorithm 5.1.1.

As already mentioned, during the reachability algorithm (see Algorithm 5.1.1), states are represented by a pair (l, z) where l is a location and z a timed zone. Given a state, the successor set is computed by the $suc_{\triangleright}$ operator, which is defined as follows:

```
1: function ForwardReach(Property φ)
2:     if (l0, z0) ⊨ φ then return YES
3:     end if
4:     Visited ← {(l0, z0)}
5:     Pending ← {(l0, z0)}
6:     while Pending ≠ ∅ do
7:         (l, z) ← next(Pending)
8:         for (l′, z′) ∈ suc▷(l, z) do
9:             if ¬IsStateAlreadyVisited?((l′, z′)) then
10:                Add((l′, z′), Pending)
11:                Add((l′, z′), Visited)
12:                if (l′, z′) ⊨ φ then return YES
13:                end if
14:            end if
15:        end for
16:    end while
17:    return NO
18: end function
```

Algorithm 5.1.1: Forward reachability algorithm.

```
1: function IsStateAlreadyVisited?((l, z))
2:     Z_l ← ⋃_{(l,z′)∈(Visited∪Pending)} z′
3:     answer ← YES
4:     if ¬IsIncludedInSet(z, Z_l) then
5:         answer ← NO
6:     end if
7:     if ∃ (l, z′) ∈ Pending_l | IsIncluded?(z′, z) then
8:         Delete((l, z′), Pending)
9:     end if
10:    return answer
11: end function
```

Algorithm 5.1.2: State already visited algorithm.

$$suc_{\triangleright}(\mathsf{l}, z) = \{(\mathsf{l}', z')/\ \langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}' \rangle \in E \wedge z' = suc_{\tau}(reset_{\alpha}(z \cap \psi)) \cap I(\mathsf{l}')\}$$

Where $reset_{\alpha}$ means putting the clocks in α to zero and $suc_{\tau}(\psi)$ means replacing the constraints of the form $x \prec c$ by $x < \infty$ while leaving the rest intact, as was presented in more detail in Section 3.1.

Section 3.2 already introduced RDBMs, a compact storage version of DBMs which store only the *Minimal Constraint Representation* of the equation system. It should be noted, however, that the having an RDBM does not automatically mean that it contains only the minimal constraints, as minimality is not preserved by all operations. This is why sometimes an explicit call to a *minimize*() operation is necessary. Algorithm 5.1.3 shows the classical pseudocode for inclusion checking in such data structure.

Algorithm 5.1.3 is trivially generalized to check if a zone is included in a set of zones (IsIncludedIn-Set?()).

```
function IsIncluded?(RDBM z1, z2)
    // z1 should be in canonical form, z2 can be minimized.
    for i = 0 to #clocks do
        for j = 0 to #clocks do
            if i == j then
                next
            end if
            if ¬(z1[i, j] ≤ z2[i, j]) then
                return NO
            end if
        end for
    end for
    return YES
end function
```

Algorithm 5.1.3: Inclusion checking algorithm.

5.2. Distributed Forward Reachability

In this section a distributed forward reachability algorithm is presented. As is common when dealing with distributed algorithms, asynchronous versions tend to minimize global waiting time, at least in principle. Usually the explanation is that processors which still have work to do can keep processing without the need to synchronize with others. This is why a distributed asynchronous architecture will be used.

The main components are:

- A *coordinator*, which is the component that starts verification, distributes workload and detects termination.
- A set of *capsules*, with the same definition as in Chapter 4. Inside them, there are:
  - An *iterator*, which makes the computation.
  - A *repository*, which stores states.
  - *Connectors*, which connects capsules.

The main algorithm is straightforward. It starts by the *coordinator* creating a part of the parallel composition of the intervening timed automata components. Because composing the whole control graph will be prohibitive in terms of time, it uses time and number-of-locations thresholds to decide how far the initial composition should proceed, leaving the rest of the task to be done on-the-fly by the *capsules* [1]. This precomposition follows no particular approach besides the mentioned thresholds and generating a subgraph of locations that are reachable (in the graph theoretic sense, not in the time-reachability sense) from the initial one.

The precomposed locations are distributed among *capsules* in a round-robin or random way but recording the assigned *capsule*.

Each *capsule* runs Algorithm 5.2.1, which is the basic forward reachability algorithm with provisions to

- forward non-local discovered states to their owning *capsule* (line 17),
- notify the *coordinator* of locations with unknown owner (line 14, the specifics are described below),
- and periodically let it known of statistics (line 23 and line 27).

[1] As mentioned in Observation 4.1, the size of the control graph is usually negligible compared to the state space.

All the *capsules* receive an initial state set that contains the initial state for one for one of them but it's empty for the rest, as dictated by the mapping kept by the *coordinator*, known as *partitioning*.

When we talk about the distributed algorithm from the perspective of a *capsule* c, a location will be *local* if it is assigned to c by the partitioning known to c at the moment and remote if it is not (that is, if P_c is the partitioning known by c, then l is local for c if $P_c[\mathsf{l}] = c$). Consequently, in a point in time, states are either local or remote depending on their location.

$Visited_c$ will be used to denote the *Visited* set at *capsule* c at a given instant. Also, $Visited_{\mathsf{l}}$ denotes the part of *Visited* (also at a given instant) that has l as location. If L is a location set, then $Visited_{\mathsf{L}}$ stands for $\bigcup_{\mathsf{l}\in\mathsf{L}} Visited_{\mathsf{l}}$.

```
1: function ForwardReachCapsule(Partition P, CapsuleNumber c, Property φ, StateSet initial)
2:     Visited ← ∅
3:     Pending ← initial
4:     found ← false
5:     while ¬GlobalEnd do
6:         while Pending ≠ ∅ ∧ ¬found do
7:             (l, z) ← next(Pending)
8:             for (l′, z′) ∈ suc▷(l, z) do
9:                 c′ ← GetOwnerCapsule(P, l′)
10:                if (l′, z′) ⊨ φ then
11:                    found ← true
12:                    NotifyCoordinator(found)
13:                else if c′ =unknown then
14:                    SendUnknownLocation(l′, COORDINATOR)
15:                    QueueUnknownState((l′, z′))
16:                else if c ≠ c′ then
17:                    SendState((l′, z′), c′)
18:                else if ¬IsStateAlreadyVisited?((l′, z′)) then
19:                    Add((l′, z′), Pending)
20:                    Add((l′, z′), Visited)
21:                end if
22:            end for
23:            if ¬GlobalEnd ∧ EnoughTimePassed() then
24:                ReportToCoordinator()
25:            end if
26:        end while
27:        if ¬GlobalEnd ∧ EnoughTimePassed() then
28:            ReportToCoordinator()
29:        end if
30:    end while
31: end function
```

Algorithm 5.2.1: Distributed forward reachability algorithm.

When we talk about the *owner of a state* (l, z), we are referring to the *capsule* where l is assigned to by the current mapping ($P_c[\mathsf{l}]$). Algorithm 5.2.1 also composes on-the-fly (at line 8), so previously unknown locations can be generated. When a new state (l, z) is computed, it can be the case that it has an *unknown owner*. I.e., that its location is not assigned to any *capsule* in the current partitioning ($P_c[\mathsf{l}] = \bot$).

States with unknown owner are queued and reported to the *coordinator* (line 14), which in turn assigns them to some *capsule* and broadcast a batch of assignments. When *capsules* receive the new partitioning they just traverse the queued states and either explore them locally or send them to their

owner *capsule*, who receive them by running Algorithm 5.2.2.

```
1: function ReceiveState(State (l, z))
2:     if ¬IsStateAlreadyVisited?((l, z)) then
3:         Add((l, z), Pending)
4:         Add((l, z), Visited)
5:     end if
6: end function
```

Algorithm 5.2.2: State reception algorithm.

We have to deal with correctness. This means proving partial correctness (either the target property is found or the state space is exhausted), non-abortion, and termination detection. The first two are dealt with in the next section, the last one in Section 5.2.2.

5.2.1. Partial Correctness & Non-Abortion

In order to prove partial correctness we will first clearly define our notion of state space in Definition 5.1. Then, by means of Observation 5.1, we will argue that if we prove partial correctness for the unreachable case, the reachable one will also be covered. We resort to that only to simplify reasoning.

Next, in Lemma 5.1 we show that the *coordinator* will not declare the end too soon. Finally, Theorem 5.1 will be proved to guarantee that, if there's no abortion (Lemma 5.1) and we are dealing with an unreachable case (Observation 5.1), then the full state space is indeed explored.

Definition 5.1 (State Space)
Given a TA $\mathcal{A}$, we call state space *to the lattice of states generated by $Reach_{fwd}(\{(\mathsf{l}_0, 0)\})$ (see page 13), being l_0 the initial location of $\mathcal{A}$.*

As was said, it will be helpful to consider the following observation:

Observation 5.1 *We call* premature cut *to the test performed in line 10 of Algorithm 5.2.1, which checks if the target property has been reached at each step (thus "premature", because the reachable state space has not yet been completely explored).*

We are dealing with a reachability algorithm that is only interested in knowing whether or not the target property is reached, and are not concerned with finding all the reachable states labeled by it.

In this case, as the test of line 10 is performed on every state found, then correctness of the algorithm is not affected by the premature cut.

■

Now let's focus on abortion. By abortion we mean the ending of the verification prior to state space exhaustion. In particular, we will deal with the threat that the *coordinator* claims global termination while there is still work to do.

Lemma 5.1 *The* coordinator *does not abort.*

Proof

The *coordinator* will not declare termination until it has received at least one report from every *capsule*. Now let's inductively suppose that at some point in time the end is not yet declared and that *capsule* c_1 is the only one active. At that time pairwise message count is zero at the *coordinator* but c_1 must have at least one pending state (otherwise the end would have been declared). To trigger

abortion c_1 should have sent a new state to some other *capsule* c_2 while reporting that it does not have any pending state to process. But this same report will unbalance *coordinator*'s message count. Even c_2's new report, which would balance it back, will be insufficient for abortion, as it will report at least one pending state (reporting an accurate count of zero pending does not configure an abortion scenario).

□

A handy lemma will be posed before dealing with complete exploration of the state space:

Lemma 5.2 *States in the Pending queue are eventually explored unless global termination is detected or the algorithm aborts.*

Proof

Trivial from Algorithm 5.2.1: only the exploration cycle of line 6 removes states from *Pending*, which has a finite number of them. This happens because of Theorem 2.1 and the fact that both Algorithm 5.2.1 and Algorithm 5.2.2 only add new states.

□

Lemma 5.2 will be used implicitly from now on. That is, is we guarantee that a state gets to the *Pending* queue, that will be as good as showing that it will be explored. This shortcut will become handy in many proofs to come.

Now, let's focus on proving that every reachable state is explored. One of the differences with the traditional algorithm (5.1.1) is that the state space exploration order changes, but it is known from [Cou78] that this is immaterial. But, how do we know that the distributed algorithm does not "skip" states? This question is formally addressed in Theorem 5.1.

Theorem 5.1 (The Distributed Algorithm Explores the Full State Space)
If there is no abortion, in an unreachable case, the full state space is explored by the distributed algorithm. I.e., every found state is explored by Algorithm 5.2.1.

Proof

We will show it by proving that every state reachable from the initial is explored by Algorithm 5.2.1.

Base case: the initial state is assigned to some *capsule* which runs Algorithm 5.2.1 on it (cf. page 51).

Inductive step: inductively suppose that (l', z') is a successor of an explored state (l, z). (l', z') can be either

1. local, in which case it will eventually be explored after being put in *Pending* (if it was not already visited),
2. remote, in which case it will be sent to another *capsule* by line 17 and then received by Algorithm 5.2.2 which will queue it into *Pending* (again, if it was not already visited), in turn leading to it being eventually explored, or
3. with unknown owner, in which case it will be queued and the *coordinator* notified. As the *coordinator* eventually replies, the state will either be queued locally or sent to its rightful owner, as was stated previously.

□

5.2.2. Termination Detection

Let's consider termination *detection*. To avoid confusion, it is convenient to clarify what detection means: we are not dealing here with the question of whether the state space is exhausted or not. Our

concern here is, provided that it is exhausted, the detection of this situation.

There are two means to detect termination.

1. When a *capsule* discovers a location with the desired property, it sends a special message to the *coordinator* (line 12 of Algorithm 5.2.1), who broadcasts the piece of news to the rest of the *capsules* and starts the distributed trace reconstruction protocol (see Section 5.2.3).
2. If at some point in time the *coordinator* infers from the *capsules*' reports (sent by lines 23 and 27 of Algorithm 5.2.1) that there are zero new states, zero pending states and there are no messages traversing the network (i.e., for every pair of *capsules* the number of messages sent and received to each other matches), then the verification has finished and the target property is unreachable.

This is a classical protocol for termination detection. Its correctness is based on pairwise message counting, and it works because no messages (of the type we are looking at) are exchanged unless there are still states being explored (line 17 of Algorithm 5.2.1).

5.2.3. Trace Reconstruction

When the target property is found it becomes necessary to generate a counter-example trace showing the sequence of discrete and timed steps that led to it from the initial state. This is easy in the monoprocessor case: each state contains a pointer to its "father" (i.e., the state that was explored to generate it) and a reference to the appropriate label. That is, if $(\mathsf{l}, z) \xrightarrow{e} (\mathsf{l}', z')$, then (l', z') has a reference to e and a pointer to (l, z).

In the distributed setting things are harder: the father state can be either local or remote. If remote, because of migrations (see Section 5.3), it can be in any *capsule* by the end of the verification. Recording the complete father state is impractical in terms of memory.

Instead of a pointer, the *location vector* of the father is recorded. The location vector is the tuple of location numbers of all the TA components (i.e., what distinguishes it from a global location, from a data structure perspective, is that it lacks the invariant and transitions).

When the target state is found, the following algorithm is run by the *coordinator*. It starts by setting a *current* variable to the target state and continues up to the obtention of the initial state.

1. Get father's location vector from *current* (call it v_f).
2. Find out who the owner *capsule* is, call it c.
3. Send *current*, the generating label (call it l) and v_f to c, requesting a state (l, z) such that it has v_f as location vector and can generate *current* through the l. I.e., such that $location\_vector(\mathsf{l}) = v_f$ and $(\mathsf{l}, z) \xrightarrow{l} current$.
4. Wait for answer.
5. Set *current* to the received state.

The actual version has some more details to handle border cases, but keeps the same general outline.

5.3. Even Workload Distribution

An even workload distribution is critical to the success of any distributed model checking tool. As our unit of distribution is the location, we need to understand what factors are involved in its workload.

Looking at Algorithm 5.2.1 we can see that the innermost operation of the cycle, and thus the most called, is IsStateAlreadyVisited?(), which in turn contains several (expensive) calls to the IsIncluded?()

zone inclusion check. So the interesting question is how many time will IsIncluded?() be called for a particular location. This is what needs to be balanced, so when we talk about *work* we are actually referring to inclusion checks.

There are none known accurate and efficient ways of responding to that question in general, but a number of approximate shortcuts can be taken, as will be explained in Section 5.3.1.

It is very likely that the workload pattern varies over time (from Chapter 4 and [Beh05]), so a rebalancing strategy is needed. Such a strategy needs to be reactive and take decisions that will have a short life span. At a given instant, each *capsule* has a number of states in its *Pending* queue. Only some of them will get to be explored before rebalancing actions need to be taken again. We don't know exactly how many, so we will resort to the concept of *immediate future* as a way to refer to the "next few states to be explored" and postpone a more precise definition until Section 5.3.1, when the details are presented.

We will start by the intuition behind our rebalancing approach and progressively introduce more detail:

1. First, figure out which are the locations that are going to be explored in the immediate future.
2. Then, obtain a numerical estimate of how much work will it take for each of them.
3. With those numbers, run a redistribution algorithm, trying to balance the quantity of work assigned to each *capsule* while at the same time shift as few locations as possible.
4. Broadcast the new mapping.
5. On reception of it, each *capsule* would start sending its $Visited_{\mathsf{l}}$ for each location l that now has a different owner.

There are many important details to consider in order to get a successful implementation of such a migration strategy. Let's start by items 1 and 2. If the redistribution algorithm is run periodically, then it can be estimated how many states will be explored before its next run, but this estimation will be time-consuming by itself. Our approach for item 1 is to look at the top n states from $Pending_{\mathsf{c}}$ for each *capsule* c, where n is a parameter that needs to be tuned.

Looking at the previously mentioned n states extracted from $Pending_{\mathsf{c}}$ we can compute how many times each location is going to be analyzed in the immediate future (understanding "immediate" as "top n"). How much work will it take each time? We don't know for sure, but for big enough *Visited* queues, the worst case will probably be close to what it is now (because some states more or less will represent only a negligible percentage of it).

So, as an estimate, multiply each location l's cardinality in the set of the top n pending states by the number of states in $Visited_{\mathsf{l}}$. This is the number we use for item 2.

The specific mechanics of our migrations algorithms are laid down in the following subsections. Being highly sophisticated, each decision is difficult to understand without its rationale, so they will be presented together with some discussion of their characteristics. Section 5.3.2 accounts for how the balancing decisions are taken, Section 5.3.3 discusses when. Finally Section 5.3.4 takes care about implementing the decided topology changes.

5.3.1. Computing Migrations

In its periodical report to the *coordinator*, each *capsule* c includes the following information (where n and m are independent run-time tunables):

- A *last repartitioning report seen* numerical stamp (see Section 5.3.4).
- *Top Pending Locations* (*TPL*): Look at the top n states from $Pending_{\mathsf{c}}$ and report tuples $\langle \mathsf{l}, k \rangle$ where l is a location and k is the number of the n top states that belong to l (i.e., k is the frequency of l in the n top states).

- *Top Visited Locs* (*TVL*): Look at $Visited_c$ and report the m locations with the greater number of states.

The *coordinator* receives these reports and checks whether enough time has passed (i.e., if a *pseudo-iteration* has finished), just storing them if not[2]. If it indeed has, reports are combined into a *General Top Pending Locations* (*GTPL*) and *General Top Visited Locs* (*GTVL*) bags. A schema is shown in Algorithm 5.3.1.

```
function ReceiveReport(CapsuleNumber c, Bag TPL, Bag TVL, Statistics S, Nat k)
    UpdateStatistics(k, c, S)
    TPL[c] ← TPL
    TVL[c] ← TVL
    if PseudoIterationFinished?() (Section 5.3.3) then
        GTPL ← ∅
        GTVL ← ∅
        for every capsule c′ do
            GTPL ← GTPL ∪ TPL[c′]
            GTVL ← GTVL ∪ TVL[c′]
        end for
        Δ ← CallParMETIS (GTPL, GTVL) (Section 5.3.2)
        BroadcastDeltas(Δ) (Section 5.3.4)
    end if
end function
```

Algorithm 5.3.1: Schema of the Rebalancing Algorithm.

Two comments are in order. A *capsule* might have failed to report within the last pseudo-iteration (although runtime tunables are in place to try to avoid this), so both general bags might contain "old" information. More importantly, what is the meaning of each bag? GTPL stands for the locations that are going to be explored during the "immediate future". In order to optimize the workload for the upcoming pseudo-iteration, these are the ones that might need balancing in the "short-term" (i.e., before the next rebalancing attempt).

On the other hand, GTVL tells (somehow) how hard it is to check inclusion for a given location l. I.e., how many states are in its $Visited_{\mathsf{l}}$ queue. It is an approximation, because only m locations are reported in each *capsules*' TVL.

Then, for each of the locations in GTPL its cardinality in GTPL is multiplied by its cardinality in GTVL (number of appearances times how many inclusion checks per each). This is the *estimated workload metric*, and is taken as the base to perform redistribution (see Section 5.3.2 for details).

It should be noticed that in this work we are migrating *visited location queues* and not *pending locations*. The rationale is that migrating the $Visited_{\mathsf{l}}$ for some location l requires sending a complete queue, which is easily extractable from its container, whereas sending $Pending_{\mathsf{l}}$ involves looping through *Pending*, eliminating states, accumulating them elsewhere, etc., which is costly. Anyway, when those states are explored they will be sent to their new owner *capsule*. Moreover, this method avoids sending a bunch of to-be-explored states to some *capsule* c, big enough so only some of them can actually be explored before a new repartitioning forces c to send them elsewhere.

5.3.2. Aiming for Even Workload

As was mentioned before, the workload metric is computed for every location l as l's cardinality in GTPL times the number of states (l, z) in GTVL.

[2]See Section 5.3.3 for a discussion on the mechanics of the *coordinator*.

$$(\forall \mathsf{l} \in \mathrm{GTPL})\ M[\mathsf{l}] = \#(\mathsf{l}, \mathrm{GTPL}) \times \sum_{(\mathsf{l},z)\in \mathrm{GTVL}} 1$$

Once the estimated workload metric is computed for each location the ParMETIS library is used to do the actual repartitioning.

ParMETIS [SKK00] is based on heuristic methods to handle efficiently graph repartitioning problems trying not only to balance weight while minimizing number of movements, but also to reach minimum cut. Although this would seem like a good idea because it minimizes communication, it is not such a pressing issue in a synchronous environment over a fast local area network. Also, we hope to get more weight balanced, less costly partitionings if minimum cut was not used.

We provide ParMETIS with a graph comprised of the locations that are present in GTPL, and the M array as their weight. To counter minimum cut we feed ParMETIS with a graph were every node is (only) connected through a zero-cost edge with a (unique) phantom, zero weight node.

After ParMETIS is done its output is compared against the current mapping of locations to *capsules*, deltas are computed, and they are broadcasted as will be explained in Section 5.3.4.

The library is capable of dealing with different weights for workload and rebalancing cost, but we consider both to be the same in our setting.

It should be pointed out that a number of good methods exist for rebalancing in the untimed scenario (see [HGGS02, NC97] among others). Unfortunately they are not directly applicable as they usually don't have to deal with locations having different (unsplittable) weights.

Next section discusses when to perform a repartitioning.

5.3.3. Opportunity

The *coordinator* is not actually an active component. It reacts on received messages. The rationale is that if nothing has changed, there's no need to act, and if something has indeed changed, some *capsule* would let it know, triggering its action.

A subtle point that some times can slip through this way of working is *elapsed time*. If pseudo-iteration take t time units, there can be a chance for a scenario where at $kt + (t - 1)$ time units the *coordinator* has enough data to redistribute workload but yet it does not, because of lack of elapsed time. Being only reactive, it may be awakened again only after a significant number of time units later. Had it had a time trigger, the repartitioning would have happened earlier.

Although the above mentioned bad scenario can indeed take place, having a time trigger opens the chance to have the processor running the *coordinator* (probably) being interrupted a number of times for no good reason.

5.3.4. Redistribution Protocol

This section deals with *implementing* the topology changes that are decided accordingly with Section 5.3.1 and Section 5.3.2. It deals with a two-sided process that is easy at one end (the *coordinator*) but quite involved at the other (the *capsules*).

Let's start by the easy part. The *coordinator* should send a *repartitioning report* (Δ, i) to every *capsule*, containing not only the deltas in the partitioning (that is, changes in the mapping from locations to *capsules*, computed as explained in the previous sections), but also a report index. This index i is a monotonically increasing positive integer.

How do *capsules* act upon reception of this report? As the process is fairly involved we will present it gradually, starting by general ideas and motivations and postponing the details for page 60.

They should update they "vision of the world" (that is, their partitioning P_c) and adapt to it. That means:

- Sending $Visited_{\mathsf{l}}$ for every location $\mathsf{l} \mid P_c[\mathsf{l}] = c \wedge (P_c + \Delta)[\mathsf{l}] \neq c$, That is, for every location that used to be local but is not anymore.
- Receiving $Visited_{\mathsf{l}}$ from other *capsules*, for now-local locations (i.e, for every location $\mathsf{l} \mid P_c[\mathsf{l}] \neq c \wedge (P_c + \Delta)[\mathsf{l}] = c$).
- Updating P_c. I.e., $P_c \leftarrow P_c + \Delta$.

It should be noted that because of the asynchronicity of the setting, there could also be a *capsule* c' that sent to-be-explored states to c that might need to be redirected to some other *capsule* c'' because by the time they reach c, c does not consider they local anymore.

Summing up the previous paragraphs, there are three kind of events that can happen in any order:

- Reception of a repartitioning report and sending of $Visited_{\mathsf{l}}$ for departing locations.
- Reception of $Visited_{\mathsf{l}}$ for arriving locations.
- Reception of to-be-explored states and re-routing to their new destination if they are not local anymore.

The last event is problematic because a careless handling could lead to messages infinitely being bounced between *capsules*. It should be noted, however, that although there are no guarantees of orderly reception of messages in general, the network does guarantee orderly delivery between pairs of nodes. Because of the above mentioned risk the following protocol is employed.

It can be conceived as a protocol to deal with the following incoming events:

- Reception of a repartitioning report (Δ, i).
- Reception of $Visited_{\mathsf{l}}$ for some locations l.
- Reception of to-be-explored states.
- Discovery (i.e., production in line 8 of Algorithm 5.2.1) of a state that can be local, remote or with unknown owner.

and then acting according to them to produce the following outputs:

- Sending $Visited_{\mathsf{l}}$ for some locations l.
- Temporary storing appart to-be-explored states until it is clear if they are local or remote.
- Adding local to-be-explored states to $Pending_{\mathsf{c}}$.
- Re-routing remote to-be-explored states to their new destination *capsule*.
- Informing the *coordinator* of states with unkown owner.

Each *capsule* keeps two indexes and a "*scratch book*". The indexes are

- k, the *last repartitioning report seen* index (*LRRS index*), and
- r, the *last repartitioning completed* index (*LRC index*).

Both start at 0 and r is always less than or equal to k. Also for each i, $r < i \leq k$ there is a "page" in the scratch book that lists the *capsules* for which a $Visited_{\mathsf{l}}$ set should have been received, and whether it should be kept local or sent to another *capsule* (i.e., it contains triplets $\langle \mathsf{l}, c, c' \rangle$). This is the invariant of the algorithm.

When a *capsule* receives information (of any kind) from another, it needs to know if the sending one is ahead or behind it, regarding the last report it received from the *coordinator*. Why is this important? Suppose *capsule* c receives a to-be-explored state (l, z) from c', but $P_c[\mathsf{l}] = c''$. What does

it mean? It can be the case that there is a new repartitioning known by c' but not by c, in which case c' is right and (l, z) should go to $Pending_{\mathsf{c}}$, or that c' is the one that was not up to date when it sent (l, z), and that state should actually be re-routed to c''.

That is why most of the incoming and outgoing events are tagged with k, the previosly mentioned last repartitioning report seen index. So the events, from the point of view of *capsule* c, become:

- Reception of a repartitioning report (Δ, i).
- Reception of $\langle Visited_{\mathsf{l}}, j\rangle$ from *capsule* c' which had j as LRRS index, when it sent it.
- Reception of to-be-explored states $\langle S, j\rangle$ where S is a state set and j was the LRRS index at the time of sending.
- Discovery (i.e., production in line 8 of Algorithm 5.2.1) of a state that can be local, remote or with unknown owner.

The outgoing events are also tagged with the LRRS index.

The full protocol will be introduced in a few paragraphs, but let's first see an sketch of it: when a *capsule* receives a repartitioning report with index i from the *coordinator*, not only P_c needs to be updated, but also it has to dispach $Visited_{\mathsf{l}}$ for locations l that are not local anymore (we call them *outgoing locations*). The problem is that, because of asynchronicity, there might be locations whose $Visited_{\mathsf{l}}$ was not yet received. How can it be sent?

The answer is to note in page i of the scratch book that such locations are due. Actually all of the locations that are now local but used to be away (we call them *incoming locations*) are written down. The only difference is whether their $Visited_{\mathsf{l}}$ should remain local or be sent away.

Up to this point, due locations are written in the scratch book. When a message containing $\langle Visited_{\mathsf{l}}, j\rangle$ arrives, page j of the scratch book is revisited, and either $Visited_{\mathsf{l}}$ is installed locally or sent away.

What happens with to-be-explored states? If a state (l, z) from $Pending_{\mathsf{c}}$ is not local –that's the easy part–, it should be sent away to $P_c[\mathsf{l}]$. If it is local we face the same scenario that if the state was discovered in some other *capsule* and sent to c. In both cases, if $Visited_{\mathsf{l}}$ is present, then it is checked for inclusion; if it is not, then (l, z) is put "on-hold", in a special-purpose queue. This queue will be revisited when $Visited_{\mathsf{l}}$ arrives.

Observation 5.2 *Please note that in the remainder of this section, the phrase "queue in Pending" should actually be understood as an abbreviation for "queue in Pending and in Visited if it is not already visited (i.e., pass it through Algorithm 5.1.2)".*

■

The full protocol is like this. Suppose *capsule* c has a *last repartitioning report seen* index k and a *last repartitioning completed* index r.

1. If it receives a new repartitioning report, it
   - *a*) updates k (safety check: the new report should have an index of $k + 1$),
   - *b*) adds a page entitled k to the scratch book (if it wasn't already there) where it takes note of the pending incoming locations (actually, the "note" is not taken if it was "scratched" before –see item 2*a*– and steps 2*d* and 2*c* are performed),
   - *c*) dispatches $\langle Visited_{\mathsf{l}}, k\rangle$ for its outgoing locations l, such that $Visited_{\mathsf{l}}$ is present,
   - *d*) notes in the scratch book the outgoing locations l (and their destination *capsule*) for which $Visited_{\mathsf{l}}$ is not yet present (those are locations which are supposed to be local but were not yet received), and
   - *e*) updates its "vision of the world" (i.e., its partitioning P_c).

As in the same report the *coordinator* informs ownership for newly discovered locations, c iterates over its special purpose queue (see item 3) and treats the states there as in item 4 or item 5 (newly discovered), depending on whether they were assigned to c or to some other *capsule*.

2. If it receives $\langle \mathit{Visited}_{\mathsf{l}}, j\rangle$ from *capsule* c', it
   a) accepts them and "scratches" c' from page j of its pending scratch book,
   b) checks if a delivery is pending (from 1d) and sends the corresponding $\mathit{Visited}_{\mathsf{l}}$ as in 1c,
   c) queues in *Pending* all the states (l, z) that were on-hold for j (see item 6), and
   d) checks if page j can be disposed and r incremented.
3. If it discovers a state (l, z) with unknown owner (i.e., l is an unknown location), it is placed in a special queue, and informed to the *coordinator* at report time.
4. If it discovers a local state (l, z), and $\mathit{Visited}_{\mathsf{l}}$ is present, it proceeds as usual, but if $\mathit{Visited}_{\mathsf{l}}$ is absent (because it was not yet received), (l, z) is put on-hold for i, being i the minimum value $r < i \leq k$ having l as pending in the scratch book.
5. If it discovers a non-local state s, it sends $\langle s, k\rangle$ to its owner *capsule* (actually, these are batched, so some $\langle S, k\rangle$ are sent).
6. If it receives to-be-explored states $\langle S, j\rangle$ from c', and
   a) if $j \leq r$, it
      1) queues in *Pending* the local ones (i.e., they are local and c has $\mathit{Visited}_{\mathsf{l}}$ for them), and
      2) re-routes the non-local ones (with k as index[3]), but
   b) if $r < j \leq k$,
      1) re-routes the non-local ones[4] (again, with k as index), and
      2) a' if j is still present in the scratch book, puts the rest of S (call it S') on-hold for j (i.e., c will only be able to explore them when $\langle \mathit{Visited}_{\mathsf{l}}, j\rangle$ arrives for all $\mathsf{l} \in \mathrm{locations}(S')$, see 2$c$),
         b' if j is not present in the scratch book but l is pending with an index less than j, it means that some other *capsule* owes $\mathit{Visited}_{\mathsf{l}}$, but with a previous index, so put them on-hold for i, being i the maximum value $r < i < j$ having l as pending in the scratch book,
         c' else, queue them in *Pending* as in item 6a1, but
   c) if $j > k$, as c' sent S knowing that all those states correspond to c at "iteration" k, they should all be put on-hold for j, as in the previous step.

Correctness needs to be proven for the improved version of the distributed protocol. Termination is postponed for Observation 5.3 and partial correctness is posed in the following theorem:

Theorem 5.2 (Migrations preserve partial correctness)
The full state space is explored by the distributed algorithm with migrations.

To prove Theorem 5.2 we need to be sure that

1. Safety: no state (l, z) is checked for inclusion without $\mathit{Visited}_{\mathsf{l}}$ present.
2. Progress: every non-local state (l, z) eventually goes into the *Pending* queue of some *capsule*.

The first one is easy: suppose *capsule* c (with indexes k_c and r_c respectively) discovers a state (l, z):

[3]This is because it is being re-routed based on "how the world looks at k". Otherwise, the receiving *capsule* will bounce it back.

[4]Actually, if $j = k$ there should be zero non-local states (i.e., both c and c' had the same "vision of world" at delivery time, and to-be-explored states are only sent to their intended destination).

- If it is not-local (step 5) it sends $\langle\{(\mathsf{l}, z)\}, k_c\rangle$ to *capsule* c'. The various relationships between k_c and $k_{c'}$ and $r_{c'}$ are covered in step 6, which handles the reception event guaranteing that it will be not be explored ahead of time (remember Observation 5.2: queuing in *Pending* covers the check for inclusion).
- If (l, z) is local, the same step 4 already checks for the presence of $Visited_{\mathsf{l}}$.
- If its owner is unkown (step 3) the *coordinator* is informed, it eventually assigns it to some *capsule* and the state is treated as a local or remote one (see last paragraph of step 1).

For the progress property, let's assume that (l, z) does not eventually get in some *Pending* queue. Then it can be the case that it is either on-hold or it is always bounced. Let's consider the on-hold case first (Lemma 5.4), with the help of the auxiliary Lemma 5.3.

Lemma 5.3 *The reception of a repartitioning report will eventually lead to* $Visited_{\mathsf{l}}$ *being sent for every outgoing location* l.

Base case: all the *capsules* start with the same partitioning, so when the first repartitioning report arrives all the outgoing locations have their *Visited* set locally.

Inductive step: let's assume that at *capsule* c all outgoing locations for the repartitioning report k have been sent and a new $k+1$ report arrives. Per inductive hypothesis all the other *capsules* will eventually send their outgoing locations for report k. As messages do not get lost all those *Visited* sets will be received. Step 2*b* guarantees that when a *Visited* set is received, pending deliveries are honored. □

Lemma 5.4 *Every state* (l, z) *that gets into a on-hold queue will eventually be taken out of it.*

In the on-hold case, (l, z) was sent to *capsule* c with and index j that was greater than r_c. If such a thing happened it was because r_c was left behind, but as the network does not loose messages, we can be sure that every *capsule* will eventually receive the repartitioning report with index j and because of Lemma 5.3, they will send their *Visited* sets. Again, as messages are not lost, they will be received by c, which will trigger steps 2*d* and 2*c*, thus taking (l, z) out of the on-hold set. □

So if there is no progress, it must be because (l, z) is being continually bounced. This means that it is always treated by the non-local part of step 6 in the receiving *capsule* c', because $j < k_{c'}$. But if it is always bouncing and never lands into a *Pending* queue, eventually all the rest of the state space will be explored except maybe for the part that starts at (l, z). This means that all *Pending* queues will become empty and therefore all the *capsules* idle. So there will be no repartitioning needs, but global termination will no be possible because there is at least one uncounted message (the one carrying (l, z)). So the topology will stabilize and (l, z) will eventually reach its final destination. □

Observation 5.3 *The termination detection (Lemma 5.1) is preserved by migrations, because these messages are also counted.*

Observation 5.4 *It is interesting to note how the redistribution protocol is actually generic, allowing to move information asynchronously and consistently among nodes in a distributed computation, while preserving required preconditions (i.e., although a piece of information might arrive requesting to be processed, that will not take place until the required background data –*$Visited_{\mathsf{l}}$ *in our case– has also arrived).*

■

Table 5.1: Experimental settings

Setting	Δt_m	$\#s$	d	s_{TPL}	$\#TVL$
M_0	10	5	20	100	100
M_1	30	1000	20	100	100
M_2	60	1000	20	100	100
M_3	60	1000	40	100	100
M_4	600	1000	20	100	100
M_5	600	1000	20	500	500

5.4. Case Studies

The technique was implemented into ZEUS, a number of case studies were selected, and a series of experiments were run in a 4 way Intel Xeon 2.66 GHz, with 2 GB of RAM, running Linux 2.6. The aim of the experiments was to answer the following questions:

- Is the migrations technique needed? Or is a static distribution settled at the beginning enough, and no rebalancing is required?
- Even if it is needed, is the migrations technique better than no migrations at all?
- Does it have an important overhead?
- Does the fragmentation phenomenon show up in the forward setting?
- Are there few locations responsible of most the work, as in backwards?
- Our technique is based on analyzing some states from *Pending* and *Visited*, taking decisions based on them, and repartition every so often. How many states should be analyzed? Should we use a frequent repartitioning or wait for longer periods?
- More specifically, as the tool has some tunable variables, is there some combination of values for them that is best suited, in general or for particular case studies?

Besides the control case with no migrations at all, six experimental settings were prepared (see Table 5.1). We varied the following options:

- Δt_m: minimum number of seconds that have to pass for the *coordinator* to compute a new migration.
- $\#s$: number of states each *capsule* has to explore before sending a new report to the *coordinator* (line 23 and line 27 of Algorithm 5.2.1).
- d: minimum accepted percentual difference between the imbalance, prior and after redistribution, for a migration set to be considered valid. I.e., before calling ParMETIS the *coordinator* computes the imbalance factor as the difference in weight between the most and less loaded *capsules*. After ParMETIS, the imbalance factor is computed again. The new one has to be at least d% better than the previous one in order to consider the migration valid. Otherwise, it is disregarded.
- s_{TPL}: the number of $Pending_{\mathsf{c}}$'s states to visit in order to construct the TPL (see Section 5.3.1).
- $\#TVL$: the number of top visited locations to include in the TVL.

In all of the runs the initial distribution was random with a fixed seed and the number of precomposed locations was set to 1000.

It should be noted that five tunable integer variables, even if they had a closed range of possible values, account for a combinatorial number of experiments, making it very hard to handle. This is why we decided to create the settings from Table 5.1. Although they do not sweep the complete spectrum of possibilities, they do give rise to some interesting issues and observations, allow to answer most of our research questions, and serve as an aim to narrow the range of each variable in the future.

The rationale for each setting is as follows:

- M_0 tries to give the *coordinator* a chance to react as fast as possible, and thus has a low $\#s$ and Δt_m.
- M_1 increases both values, trying to counter the overhead of frequent transmissions and repartitionings attempts.
- M_2 tries to diminish the number of repartitionings to counter the possible overhead of moving state queues around.
- Doubling d, M_3 takes more conservative decisions about which migrations are worth doing.
- M_4 is aimed at long case studies, trying to see the effect of a long delay between reports.
- Lastly, M_5, which increases s_{TPL} and $\#TVL$ aims at the same long cases, this time trying to feed more information into the heavy location detection mechanism.

The results for the control setting are presented in Table 5.2. Along with the time and speedup for each number of processors a primed version is presented. It represents time and speedup without counting TA copying and trace generation. Copying involves resorting to external tools[5] that might have an important time variability. We choose to leave this out in order to consider only the efficiency of the algorithms we developed.

Table 5.2: Control setting (no migrations) – total time (seconds)

Example	t_1 (t'_1)	t_2 (t'_2) speedups	t_3 (t'_3) speedups	t_4 (t'_4) speedups
RCS5	31 (21)	42 (22)	34 (6)	41 (12)
true		0.74 (0.95)	0.91 (3.50)	0.76 (1.75)
RCS5	832 (832)	407 (405)	424 (423)	309 (306)
false		2.04 (2.05)	1.96 (1.97)	2.69 (2.72)
FDDI4	15 (1)	26 (5)	28 (2)	43 (5)
true		0.58 (0.20)	0.54 (0.50)	0.35 (0.20)
FDDI4	20 (20)	20 (13)	22 (7)	34 (9)
false		1.00 (1.54)	0.91 (2.86)	0.59 (2.22)
FDDI8	397 (372)	308 (246)	233 (160)	236 (128)
true		1.29 (1.51)	1.70 (2.33)	1.68 (2.91)
FDDI8	6 254 (6254)	3 674 (3 672)	2 879 (2 876)	3 110 (3 102)
false		1.70 (1.70)	2.17 (2.17)	2.01 (2.02)
Conv6ABC	42 (28)	45 (20)	39 (8)	113 (41)
true		0.93 (1.40)	1.08 (3.50)	0.37 (0.68)
Conv6ABC	3 241 (3241)	1 851 (1 848)	1 570 (1 565)	1 234 (1 224)
false		1.75 (1.75)	2.06 (2.07)	2.63 (2.65)
Pipe6'	114 (82)	124 (35)	147 (7)	106 (22)
true		0.92 (2.34)	0.78 (11.71)	1.08 (3.73)
Pipe6'	42 (42)	41 (39)	49 (46)	43 (39)
false		1.02 (1.08)	0.86 (0.91)	0.98 (1.08)
MinePump	2 328 (2238)	1 035 (941)	1 483 (331)	1 721 (364)
true		2.25 (2.38)	1.57 (6.76)	1.35 (6.15)
MinePump	17 013 (17013)	31 653 (31 652)	11 097 (11 095)	15 650 (15 647)
false		0.54 (0.54)	1.53 (1.53)	1.09 (1.09)
MinePump'	83 (37)	125 (32)	80 (29)	100 (13)
true		0.66 (1.16)	1.04 (1.28)	0.83 (2.85)
MinePump'	446 (446)	841 (840)	694 (692)	489 (486)
false		0.53 (0.53)	0.64 (0.64)	0.91 (0.92)

[5]Currently, `scp`, the secure remote copy program, is used.

Trace generation only affects reachable cases, uses the algorithm presented in Section 5.2.3 and can be thought of as a "constant" delay at the end of the verification. It is not actually constant, but the variation comes from the length of the particular trace being found and the states explored so far, and not directly from the distribution itself.

When reporting speedup and relative performance of different settings, the time without copying and trace generation will be used. When dealing with the percentage of time dedicated to I/O, etc. (for example, Table 5.3), only the copying will be disregarded, because all the other counters are still active during trace reconstruction.

Table 5.2 has good and bad news. On one hand some case studies (e.g., *FDDI8* false) do benefit from the distribution, cutting their running time in half with two processors. On the other, the results are irregular, with a speedup far from linear and some cases (like *MinePump* false with two processors) suffer an intolerable overhead.

On what follows, we will analyze case studies that have a monoprocessor runtime of more than two minutes. In the ones that take less, it is very easy for the overhead to surpass any efficiencies from distribution.

Table 5.3 breaks the time into interesting classes for analysis. All the columns show percentages of total time. The second column shows the sum of idle times (which happen when a *capsule* has an empty *Pending* queue), along with the idle time of each *capsule* in descending order. Ideally, both the total idle time and the different between each *capsules*'s idle time should be low. An important difference between *capsule* idle times shows unbalanced load. Then there is I/O time, which is partly included in the idle column (because *capsules* might be doing pending I/O while idle) and partly included into the next column, which represents serialization time. This last timer computes the time spent *serializing* (preparing states to be sent over the network) and *deserializing* (turning byte streams back into states). Because some parts of the process are really tied to networking, the I/O timer is partly included into the serialization one.

Although omitted in the table, the first interesting results is that reaching the 1000 precomposed locations took at most 1 second in all the runs, validating the claim that precomposing would have a negligible overhead.

Table 5.3 does indeed show large fragments of idle time, which are coherent with the poor speedups seen in Table 5.2. There is also imbalance among *capsules* (e.g., *MinePump* false, *RCS5* false), justifying the need for a rebalancing mechanism.

Table 5.3: Metrics for the control setting (I/O is mostly included in the serialization column and part in idle).

Example/#proc	% idle (each *capsule*)	% I/O	% serialization
RCS5 false / 1	0.00 (0)	0.48	0.00
RCS5 false / 2	6.79 (11, 2)	3.70	43.46
RCS5 false / 3	12.29 (22, 10, 4)	6.07	51.62
RCS5 false / 4	21.65 (31, 30, 21, 4)	8.66	54.33
FDDI8 true / 1	0.00 (0)	6.05	0.00
FDDI8 true / 2	0.67 (1, 0)	14.72	2.68
FDDI8 true / 3	0.16 (0, 0, 0)	20.40	2.49
FDDI8 true / 4	1.59 (2, 0, 0, 0)	31.75	3.57
FDDI8 false / 1	0.00 (0)	0.00	0.00
FDDI8 false / 2	0.05 (0, 0)	0.25	5.58
FDDI8 false / 3	0.56 (2, 0, 0)	0.65	8.91
FDDI8 false / 4	0.10 (0, 0, 0, 0)	0.45	10.92
Conv6ABC false / 1	0.00 (0)	0.19	0.00
Conv6ABC false / 2	1.06 (2, 0)	1.16	30.57
Conv6ABC false / 3	4.13 (9, 2, 2)	1.02	41.64
Conv6ABC false / 4	7.37 (10, 10, 8, 1)	1.98	45.96
Pipe6' true / 1	0.00 (0)	28.07	0.00
Pipe6' true / 2	4.02 (4, 4)	44.20	7.14
Pipe6' true / 3	0.53 (1, 0, 0)	41.01	10.58
Pipe6' true / 4	5.94 (8, 6, 5, 0)	59.69	10.00
MinePump true / 1	0.00 (0)	3.14	0.00
MinePump true / 2	20.59 (41, 0)	19.66	23.46
MinePump true / 3	20.54 (61, 0, 0)	32.10	31.38
MinePump true / 4	13.82 (35, 11, 8, 2)	41.95	29.15
MinePump false / 1	0.00 (0)	0.06	0.00
MinePump false / 2	19.17 (38, 0)	9.38	36.59
MinePump false / 3	41.96 (85, 41, 0)	21.77	37.25
MinePump false / 4	54.25 (84, 75, 58, 0)	22.87	41.04
MinePump' true / 1	0.00 (0)	54.22	0.00
MinePump' true / 2	5.04 (7, 3)	48.74	9.24
MinePump' true / 3	8.44 (15, 9, 1)	44.44	14.67
MinePump' true / 4	16.33 (32, 24, 5, 4)	43.11	20.41
MinePump' false / 1	0.00 (0)	0.00	0.00
MinePump' false / 2	40.30 (80, 1)	25.83	26.55
MinePump' false / 3	44.89 (71, 64, 0)	23.99	31.02
MinePump' false / 4	49.59 (83, 58, 56, 1)	22.07	27.67

Table 5.4: Metrics for the M_0 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	37.22 (74, 0)	0.26	22.43	17.28
RCS5 false / 3	39.91 (59, 2, 0)	0.35	23.16	23.95
RCS5 false / 4	49.05 (72, 67, 53, 4)	0.22	21.70	37.76
FDDI8 true / 2	1.38 (3, 0)	0.20	16.80	1.38
FDDI8 true / 3	3.00 (5, 4, 0)	0.00	36.64	2.25
FDDI8 true / 4	4.19 (7, 5, 4, 2)	0.00	42.26	2.28
FDDI8 false / 2	4.16 (8, 0)	0.07	2.81	6.35
FDDI8 false / 3	0.65 (1, 1, 0)	0.10	2.29	8.07
FDDI8 false / 4	4.87 (17, 1, 1, 0)	0.13	3.33	10.35
Conv6ABC false / 2	9.10 (18, 0)	0.45	1.95	16.59
Conv6ABC false / 3	19.57 (29, 28, 2)	0.31	4.15	26.04
Conv6ABC false / 4	14.82 (26, 20, 13, 1)	0.36	3.42	37.46
Pipe6' true / 2	7.84 (12, 4)	0.00	50.00	5.88
Pipe6' true / 3	10.69 (32, 0, 0)	0.23	40.36	5.17
Pipe6' true / 4	6.54 (8, 6, 0, 0)	0.38	49.23	6.54
MinePump true / 2	0.32 (0, 0)	0.27	6.16	29.26
MinePump true / 3	1.89 (5, 0, 0)	0.23	51.66	10.60
MinePump true / 4	2.81 (9, 1, 1, 0)	0.09	59.51	11.28
MinePump false / 2	13.18 (27, 1)	0.09	7.72	35.21
MinePump false / 3	14.89 (34, 9, 2)	0.07	6.82	42.26
MinePump false / 4	38.13 (73, 69, 11, 0)	0.03	16.05	41.08
MinePump' true / 2	9.21 (13, 5)	0.00	59.21	5.26
MinePump' true / 3	5.02 (14, 1, 0)	0.72	53.05	15.77
MinePump' true / 4	7.87 (15, 7, 6, 4)	0.00	47.19	17.13
MinePump' false / 2	6.24 (10, 3)	0.30	3.72	36.02
MinePump' false / 3	25.58 (56, 20, 0)	0.00	13.91	36.37
MinePump' false / 4	44.31 (67, 56, 53, 1)	0.17	19.47	39.11

Table 5.5: Metrics for the M_1 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	20.62 (38, 4)	0.60	15.23	26.26
RCS5 false / 3	27.00 (45, 33, 3)	0.32	12.98	44.97
RCS5 false / 4	32.69 (57, 48, 17, 9)	0.16	12.42	47.13
FDDI8 true / 2	4.05 (6, 2)	1.22	14.19	4.05
FDDI8 true / 3	10.72 (18, 12, 3)	1.28	22.26	4.43
FDDI8 true / 4	1.65 (2, 2, 1, 0)	0.55	34.25	3.08
FDDI8 false / 2	0.51 (1, 0)	0.61	0.54	6.44
FDDI8 false / 3	3.55 (10, 0, 0)	0.31	3.49	8.41
FDDI8 false / 4	3.00 (5, 3, 3, 2)	0.25	3.00	11.72
Conv6ABC false / 2	7.26 (14, 1)	0.15	1.45	32.66
Conv6ABC false / 3	2.75 (4, 3, 0)	0.07	1.39	40.89
Conv6ABC false / 4	17.97 (41, 20, 7, 4)	0.11	3.57	46.75
Pipe6' true / 2	7.11 (10, 4)	0.00	36.93	5.28
Pipe6' true / 3	16.57 (47, 3, 0)	0.19	43.13	5.99
Pipe6' true / 4	7.87 (9, 7, 0, 0)	0.00	57.41	7.87
MinePump true / 2	7.54 (15, 0)	0.06	14.40	28.62
MinePump true / 3	12.13 (36, 0, 0)	0.00	23.42	34.93
MinePump true / 4	8.20 (16, 13, 2, 1)	0.11	21.29	37.42
MinePump false / 2	14.35 (28, 1)	0.08	7.52	37.26
MinePump false / 3	17.36 (28, 23, 2)	0.14	8.58	42.97
MinePump false / 4	34.82 (73, 58, 7, 0)	0.04	15.60	37.30
MinePump' true / 2	7.58 (9, 6)	0.00	71.97	5.30
MinePump' true / 3	1.72 (3, 2, 0)	0.00	64.94	9.77
MinePump' true / 4	29.98 (63, 52, 3, 2)	0.00	40.38	23.08
MinePump' false / 2	28.38 (56, 1)	0.33	17.74	19.88
MinePump' false / 3	29.71 (56, 28, 5)	0.10	15.39	37.35
MinePump' false / 4	40.69 (66, 63, 30, 5)	0.07	18.11	39.08

Table 5.6: Metrics for the M_2 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	27.28 (54, 1)	0.12	18.03	19.11
RCS5 false / 3	16.82 (46, 4, 0)	0.00	8.15	49.13
RCS5 false / 4	53.79 (89, 75, 50, 1)	0.12	23.74	36.72
FDDI8 true / 2	14.94 (27, 3)	1.57	18.80	3.86
FDDI8 true / 3	10.80 (16, 15, 1)	1.29	22.18	4.81
FDDI8 true / 4	1.29 (2, 0, 0, 0)	0.00	32.33	4.20
FDDI8 false / 2	3.30 (5, 1)	0.35	2.07	7.50
FDDI8 false / 3	0.58 (2, 0, 0)	0.36	0.56	9.77
FDDI8 false / 4	3.34 (5, 4, 4, 0)	0.37	2.57	10.33
Conv6ABC false / 2	5.89 (10, 2)	0.11	1.15	29.30
Conv6ABC false / 3	10.60 (21, 10, 0)	0.11	2.06	42.82
Conv6ABC false / 4	8.58 (16, 12, 5, 1)	0.09	1.83	44.17
Pipe6' true / 2	4.25 (5, 4)	0.00	36.79	6.60
Pipe6' true / 3	1.45 (2, 0, 0)	0.00	58.70	8.70
Pipe6' true / 4	15.23 (55, 4, 1, 1)	0.04	53.71	7.10
MinePump true / 2	1.09 (1, 1)	0.00	20.27	28.40
MinePump true / 3	22.03 (66, 0, 0)	0.02	24.78	29.57
MinePump true / 4	11.67 (28, 16, 2, 1)	0.08	26.08	38.08
MinePump false / 2	1.48 (3, 0)	0.01	0.94	28.46
MinePump false / 3	13.22 (25, 14, 0)	0.01	6.76	35.20
MinePump false / 4	37.54 (67, 54, 28, 1)	0.02	15.16	50.46
MinePump' true / 2	6.34 (7, 6)	0.00	65.49	8.45
MinePump' true / 3	3.09 (7, 2, 0)	0.00	64.81	11.73
MinePump' true / 4	12.42 (25, 17, 5, 3)	0.00	52.15	19.21
MinePump' false / 2	10.48 (20, 1)	0.00	5.29	31.54
MinePump' false / 3	33.66 (67, 32, 2)	0.08	15.43	39.19
MinePump' false / 4	43.67 (77, 68, 28, 2)	0.16	18.98	38.98

Table 5.7: Metrics for the M_3 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	14.86 (25, 5)	1.86	7.43	33.86
RCS5 false / 3	22.06 (36, 24, 6)	0.00	11.84	40.69
RCS5 false / 4	41.39 (63, 51, 49, 2)	0.41	17.38	46.36
FDDI8 true / 2	13.26 (25, 2)	1.26	18.18	4.17
FDDI8 true / 3	0.64 (1, 0, 0)	1.41	21.79	3.46
FDDI8 true / 4	14.27 (28, 26, 2, 0)	0.61	33.40	5.16
FDDI8 false / 2	1.22 (2, 0)	0.60	0.68	6.66
FDDI8 false / 3	0.63 (2, 0, 0)	0.32	0.59	9.06
FDDI8 false / 4	4.26 (7, 4, 4, 3)	0.23	4.60	10.56
Conv6ABC false / 2	10.33 (20, 1)	0.10	2.68	26.56
Conv6ABC false / 3	6.23 (11, 4, 4)	0.02	1.57	41.58
Conv6ABC false / 4	21.70 (51, 13, 13, 10)	0.07	4.15	43.46
Pipe6' true / 2	11.33 (20, 3)	0.39	46.09	3.52
Pipe6' true / 3	0.81 (2, 1, 0)	0.27	39.25	9.95
Pipe6' true / 4	6.54 (8, 6, 0, 0)	0.00	53.08	9.62
MinePump true / 2	1.09 (1, 1)	0.00	20.27	28.40
MinePump true / 3	22.03 (66, 0, 0)	0.02	24.78	29.57
MinePump true / 4	11.67 (28, 16, 2, 1)	0.08	26.08	38.08
MinePump false / 2	1.48 (3, 0)	0.01	0.94	28.46
MinePump false / 3	13.22 (25, 14, 0)	0.01	6.76	35.20
MinePump false / 4	37.54 (67, 54, 28, 1)	0.02	15.16	50.46
MinePump' true / 2	6.58 (8, 5)	2.63	57.24	5.92
MinePump' true / 3	2.38 (5, 2, 0)	0.60	59.52	12.50
MinePump' true / 4	17.39 (33, 24, 9, 4)	0.27	48.10	18.48
MinePump' false / 2	2.82 (5, 1)	0.00	2.07	30.60
MinePump' false / 3	35.13 (57, 48, 0)	0.00	17.92	32.80
MinePump' false / 4	45.19 (68, 56, 54, 2)	0.00	17.89	38.14

Table 5.8: Metrics for the M_4 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	9.20 (16, 3)	0.11	5.63	35.40
RCS5 false / 3	60.05 (90, 90, 0)	0.00	39.75	15.92
RCS5 false / 4	24.61 (41, 33, 16, 9)	0.15	10.96	56.94
FDDI8 true / 2	1.63 (2, 1)	1.63	14.50	3.25
FDDI8 true / 3	0.62 (1, 0, 0)	0.62	18.70	4.92
FDDI8 true / 4	1.47 (2, 2, 0, 0)	1.13	28.51	4.86
FDDI8 false / 2	17.37 (35, 0)	0.34	11.78	5.69
FDDI8 false / 3	13.74 (41, 0, 0)	0.39	9.75	8.67
FDDI8 false / 4	0.14 (0, 0, 0, 0)	0.16	0.69	11.95
Conv6ABC false / 2	6.24 (7, 5)	0.17	1.87	34.16
Conv6ABC false / 3	6.28 (10, 8, 1)	0.02	2.32	42.48
Conv6ABC false / 4	15.37 (34, 16, 11, 0)	0.18	2.37	41.87
Pipe6' true / 2	1.63 (2, 1)	0.18	42.21	3.62
Pipe6' true / 3	0.65 (1, 0, 0)	0.00	41.42	5.83
Pipe6' true / 4	7.29 (11, 7, 6, 0)	0.00	50.00	9.38
MinePump true / 2	1.58 (2, 0)	0.00	20.02	25.90
MinePump true / 3	19.59 (46, 13, 0)	0.00	35.01	25.93
MinePump true / 4	24.14 (55, 35, 6, 0)	0.02	18.67	45.48
MinePump false / 2	10.20 (20, 1)	0.01	5.39	38.13
MinePump false / 3	16.57 (30, 14, 5)	0.01	6.43	50.35
MinePump false / 4	36.96 (69, 63, 14, 2)	0.01	16.61	40.91
MinePump' true / 2	7.33 (9, 5)	0.00	59.33	6.67
MinePump' true / 3	1.92 (2, 0, 0)	0.00	63.46	14.10
MinePump' true / 4	13.54 (22, 19, 7, 6)	0.35	54.51	13.89
MinePump' false / 2	27.89 (55, 1)	0.00	14.82	27.97
MinePump' false / 3	35.86 (65, 41, 1)	0.00	19.23	36.05
MinePump' false / 4	55.83 (90, 76, 57, 1)	0.08	26.74	30.58

Table 5.9: Metrics for the M_5 setting.

Example/#proc	% idle (each *capsule*)	% migration	% I/O	% serialization
RCS5 false / 2	9.95 (20, 0)	0.13	5.57	32.10
RCS5 false / 3	13.57 (20, 20, 0)	0.00	5.57	53.01
RCS5 false / 4	24.73 (42, 28, 26, 3)	0.45	10.18	52.95
FDDI8 true / 2	1.23 (1, 1)	0.68	15.53	3.95
FDDI8 true / 3	1.07 (3, 0, 0)	1.22	35.62	8.60
FDDI8 true / 4	2.36 (5, 2, 1, 0)	0.32	22.96	6.65
FDDI8 false / 2	1.07 (2, 0)	0.69	0.61	6.60
FDDI8 false / 3	0.15 (0, 0, 0)	0.50	0.89	8.12
FDDI8 false / 4	0.21 (0, 0, 0, 0)	0.28	1.32	10.32
Conv6ABC false / 2	3.75 (7, 0)	0.02	0.81	29.17
Conv6ABC false / 3	49.01 (74, 73, 0)	0.11	25.30	24.39
Conv6ABC false / 4	36.85 (56, 46, 45, 0)	0.15	9.44	34.19
Pipe6' true / 2	3.19 (4, 3)	1.06	35.82	4.96
Pipe6' true / 3	0.44 (1, 0, 0)	0.89	48.44	6.67
Pipe6' true / 4	7.81 (9, 8, 6, 0)	1.56	58.59	7.03
MinePump true / 2	11.08 (22, 0)	0.02	8.12	36.54
MinePump true / 3	14.52 (23, 21, 0)	0.11	15.98	43.71
MinePump true / 4	16.45 (53, 7, 3, 3)	0.00	59.38	18.26
MinePump false / 2	8.73 (15, 2)	0.00	4.01	36.66
MinePump false / 3	33.30 (52, 48, 0)	0.01	14.68	46.40
MinePump false / 4	34.96 (86, 32, 19, 3)	0.01	15.08	45.91
MinePump' true / 2	6.00 (7, 5)	0.00	62.67	9.33
MinePump' true / 3	5.56 (13, 3, 0)	0.00	62.78	8.33
MinePump' true / 4	7.50 (9, 7, 6, 0)	0.28	43.06	23.61
MinePump' false / 2	7.64 (12, 3)	0.00	4.03	30.57
MinePump' false / 3	26.02 (72, 5, 2)	0.00	13.20	37.44
MinePump' false / 4	38.97 (69, 53, 33, 2)	0.00	17.22	40.18

Table 5.10: Heavy locations.

Example	#locs	#states	#incl checks/loc avg ± stddev	#locs over avg #locs (%)	% checks heavy
RCS5 false	1 617	277 009	1 583 864 ± 15 548 417	142 (8.00)	95.88
FDDI8 true	4 113	8 596	113 ± 151	1 332 (32.00)	85.09
FDDI8 false	15 104	136 775	50 489 ± 179 967	1 977 (13.00)	94.10
Conv6ABC false	31 443	1 369 123	1 467 277 ± 27 662 678	958 (3.00)	98.60
Pipe6' true	1 614	52 999	7 020 ± 23 358	265 (16.00)	88.54
MinePump true	1 425	387 150	58 410 980 ± 704 846 470	74 (5.00)	97.79
MinePump false	1 428	1 172 435	$2\ 371 \cdot 10^6 \pm 25\ 250 \cdot 10^6$	76 (5.00)	97.42
MinePump' true	677	31 652	39 907 ± 268 415	43 (6.00)	96.34
MinePump' false	677	131 384	5 771 325 ± 43 773 525	38 (5.00)	96.97

Table 5.10 analyzes the concept of heavy locations, previously analyzed for the backwards algorithm in Section 4.5.1, this time in the forward setting. For each case study, it shows the number or reachable locations, states, and the average and standard deviation of the number of inclusion checks required per location. Next column shows the number of locations that are above average. These are the ones we call *heavy*. Last column shows how much work (inclusion checks) they are responsible for. The immense standard deviations should be noted, showing that the work required varies greatly among locations. Also interesting, the small percentage of heavy ones, and the big amount of the total work they contribute.

Although the number of heavy locations is small, it is still superior to the number of *capsules*. Would a fair, static partitioning, balance the load? Exploring this question Table 5.11 shows how, at different sample points during the runs, the locations responsible for most of the work are not the same. We used a period of 60 seconds for the samples.

For each case study, Table 5.11 has as first column the period (sequential) number and in parenthesis the percentage of work (inclusion checks) that happened during that period. The top most significant periods are shown. Second column features average and standard deviation of the number of checks per location during the period. The third has the number of location requiring more work than the average of the period, over the number of locations checked during the period and, in parenthesis, this fraction as a percentage. These are the locations we call *heavy for the period*. Last column represents the number of locations that were heavy both in the current and (sequentially) previous period, and how much (in percentage) they represent from the total number of current heavy.

We can see that in each period the number of heavy locations is small (consistently with what happens globally). It is interesting to note how the case studies differentiate in two groups:

1. either the top periods show a small contribution, suggesting an even distribution of workload among periods, and the heavy locations are –although not matching in full– mostly the same ones that in the previous periods, suggesting some degree of stability, (most case studies) or
2. the top periods contribute the most of the work, but the variability is important (*FDDI8* true).

What can be concluded from Table 5.11 is that there is justification for rebalancing on-the-fly, although how often to do it is not that clear and seems to depend on each case study.

Fig. 5.1 and Fig. 5.2 show the speedups obtained using the different migration settings plus the scenario with no migrations at all. A few things can be concluded from the charts.

1. There's no clear winner: there is no scenario that performs substantially better than the rest.
2. Reachable case studies show superlinear speedups. This happens because reachable cases end as soon as a target location is reached. If some distribution assigns them to a less loaded *capsule*, their chances of being discovered sooner increase.
3. Apart from that, performance is far from linear in general.
4. Nevertheless, some some cases do exhibit good speedups: *FDDI8* false, *Conveyor6A* false with M_2 or M_4, *RCS5* false with M_5.
5. Promisingly, in all case studies but *MinePump* false with 3 *capsules*, there is always a migration setting that outperforms the no-migrations scenario. In many cases the difference is notable (for instance, *MinePump*' false or *FDDI8* false).
6. In only few cases M_3 is better than M_2 (they only varied in d), showing that there is no need to be highly conservative in the value of that threshold.
7. M_5, the setting that examines much more locations from the *Visited* and *Pending* queues in order to compute migrations, tends to be among the top performers with an increased number of *capsules* (of course, this should be confirmed with a larger number of processors).
8. M_0, which reports and takes decisions frequently, seems to be a good performer on reachable cases, with notable exceptions like *MinePump*'.

9. M_1 is usually outperformed by M_2, probably because of the lower reporting overhead that differentiates them.

In Section 5.5 we will analyze ways to improve our results. Let's turn our attention to Tables 5.4 through 5.9, which report the same metrics that Table 5.3 did for the control case, but for the M_0 to M_5 settings.

The new column in the tables accounts for time devoted to migrations. It should be pointed out that the timer includes serialization time but not I/O. However, the increased I/O could be seen in the corresponding column.

Looking at both column over all the tables it can be concluded that migrations do not impose a great overhead *per se*, but rather can change the load pattern of the *capsules*.

Usually, better speedups correlate with less idle time. Unluckily there is still more important information to collect about them. Not only the total and per *capsule* idle time matter, but also *when* they occur. For a given number of idle seconds, it is better that two *capsules* idle more or less at the same time than that they "take turns", effectively doubling total time.

This "turning" phenomenon, which is more likely to happen if only a few locations generate most of the work, could be the explanation for pathological case studies like *MinePump* false. For instance, its metrics for M_1 and 2 *capsules* (in Table 5.5) are much better than in the control setting (Table 5.3) and yet its timing is worst. According to Table 5.10, it indeed has only 5% of heavy locations. In an asynchronous environment like ours the type of timing information that would confirm the "turning" is very difficult to obtain and thus we cannot assert certainly that this is the explanation.

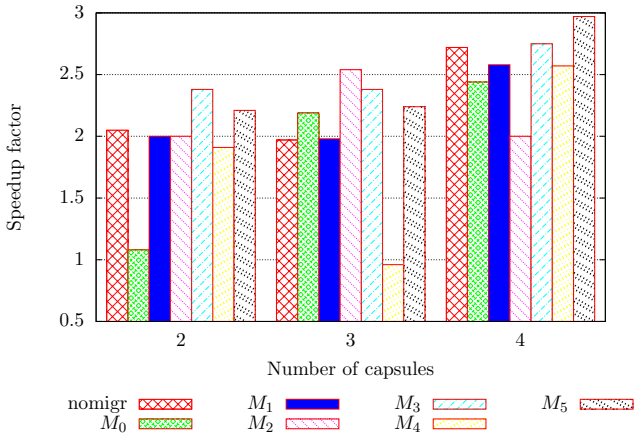
Another possible cause is *fragmentation*. The difference in the order of explored states leads to more states being found. This seems to be happening, at least in *MinePump* false, as shown in Table 5.12. This table selects one migration setting (M_1) which performs worse than not using migrations, and compares the number of inclusion checks required.

As can be seen in Table 5.12 and Fig. 5.2(b), there is some correlation between speedups and the number of inclusion checks. It also seems that some increase in the number of checks is compensated by the extra processors to handle it.

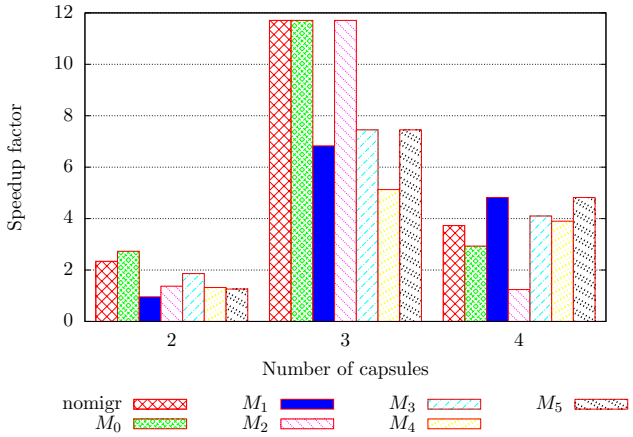
The same phenomenon was also reported in [Beh05], for a distributed version of the forward algorithm. They used the following heuristic to diminish the effects of the problem: order the states in *Pending* according to their distance from the initial state, trying to obtain an approximation of breadth first search. We should explore this technique in the future. In particular, to understand which kind of *distance* works well: hamming distance, control-graph shortest path, timed distance, etc.

It should be clear that not all the case studies suffer from fragmentation in the same manner, or at all. Take for instance, *RCS5* false, which performs reasonably well on M_5 and with dips on M_4. As can be seen in Table 5.13, there is no linear correlation between inclusion checks and speedup, featuring even a decrease in the number of inclusion checks in M_4 and M_5 with 3 *capsules*. This suggests that the other factors were more influential than fragmentation for *RCS5*.

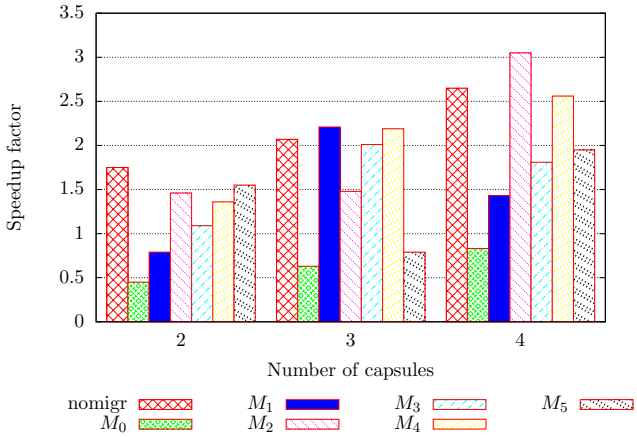
To fully understand how fragmentation affects the distributed forward algorithm more metrics should be collected during the distributed run (such as number of *different* states generated by each *capsule*, number of repetitions, etc.). This involves extra technical difficulties (for instance, avoiding that the delay imposed by the collection mechanisms alter the workload pattern, etc.) and properly implementing migrations was challenging enough. Thus, we plan to explore this specific issue in future research work.



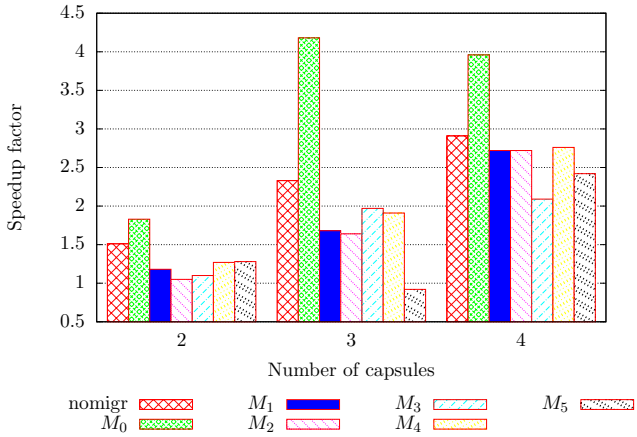
(a) *RCS5* false.



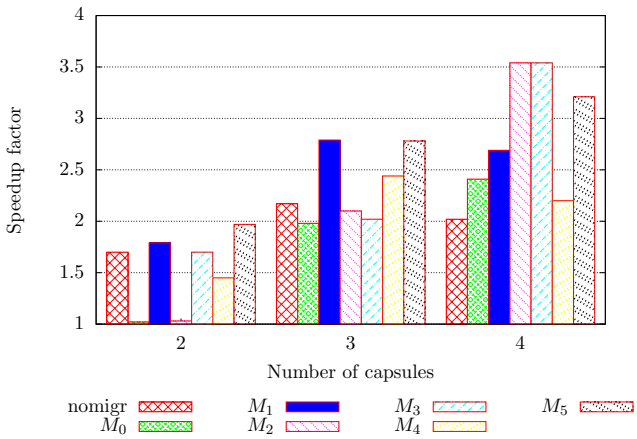
(b) *Pipe6'* true.



(c) *Conv6ABC* false.

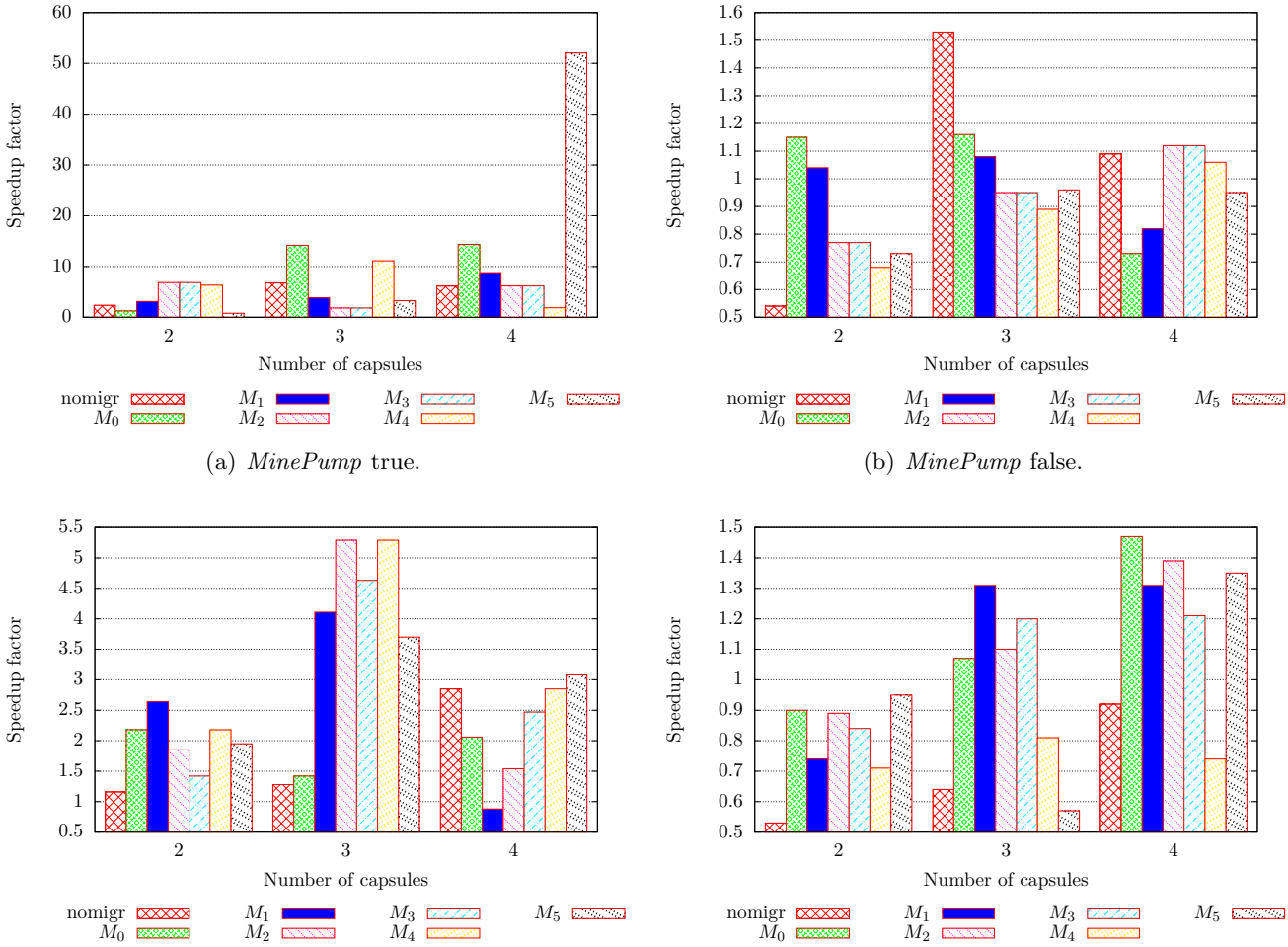


(d) *FDDI8* true.



(e) *FDDI8* false.

Figure 5.1: Speedups for different migration settings.



(a) *MinePump* true.

(b) *MinePump* false.

(c) *MinePump*' true.

(d) *MinePump*' false.

Figure 5.2: Speedups for different migration settings (cont.).

Table 5.11: Heavy locations per period.

Example / period nbr. (contribution %)	#incl checks/loc avg ± stddev	#locs over avg #locs/total (%)	#locs over avg in previous #locs (%)
RCS5 false			
8 (11.01)	93 568.21 ± 828 607.32	33 / 372 (8.00)	20 (60.61)
7 (10.28)	76 613.09 ± 323 643.76	45 / 424 (10.00)	19 (42.22)
2 (9.42)	48 562.89 ± 204 803.66	60 / 613 (9.00)	23 (38.33)
6 (9.29)	58 396.39 ± 436 362.19	62 / 503 (12.00)	30 (48.39)
FDDI8 true			
7 (39.49)	66.05 ± 52.89	541 / 1 311 (41.00)	35 (6.47)
5 (16.29)	41.99 ± 30.41	371 / 851 (43.00)	36 (9.70)
6 (14.11)	51.32 ± 37.17	319 / 603 (52.00)	140 (43.89)
4 (12.48)	32.21 ± 21.65	446 / 850 (52.00)	106 (23.77)
FDDI8 false			
85 (5.11)	870.12 ± 1 213.50	263 / 1 021 (25.00)	191 (72.62)
83 (3.51)	759.76 ± 946.47	266 / 803 (33.00)	117 (43.98)
84 (3.45)	817.49 ± 1 115.46	248 / 733 (33.00)	145 (58.47)
82 (3.23)	821.39 ± 1 000.51	226 / 684 (33.00)	0 (0.00)
Conv6ABC false			
43 (7.02)	46 541.72 ± 216 863.71	140 / 1 201 (11.00)	60 (42.86)
39 (3.64)	45 161.79 ± 251 782.75	57 / 642 (8.00)	53 (92.98)
37 (3.62)	40 122.50 ± 241 702.02	76 / 718 (10.00)	64 (84.21)
38 (3.59)	32 941.35 ± 220 469.17	63 / 867 (7.00)	55 (87.30)
Pipe6' true			
1 (100.00)	4 153.23 ± 13 579.42	385 / 2 190 (17.00)	0 (0.00)
MinePump true			
22 (3.42)	214 956.13 ± 951 317.81	23 / 179 (12.00)	12 (52.17)
25 (3.40)	200 430.12 ± 862 196.86	24 / 191 (12.00)	11 (45.83)
23 (3.38)	177 898.95 ± 733 189.22	31 / 214 (14.00)	14 (45.16)
20 (3.30)	144 911.24 ± 708 030.75	37 / 256 (14.00)	18 (48.65)
MinePump false			
263 (0.68)	404 472.17 ± 1 323 133.93	26 / 164 (15.00)	15 (57.69)
181 (0.46)	739 479.53 ± 2 817 666.03	6 / 60 (10.00)	6 (100.00)
145 (0.44)	97 518.08 ± 712 090.75	38 / 442 (8.00)	29 (76.32)
204 (0.44)	575 978.15 ± 1 588 027.85	11 / 74 (14.00)	7 (63.64)
MinePump' true			
1 (100.00)	20 343.31 ± 132 483.25	43 / 621 (6.00)	0 (0.00)
MinePump' false			
6 (23.53)	110 126.20 ± 577 041.63	32 / 414 (7.00)	31 (96.88)
4 (19.61)	164 462.58 ± 746 501.09	26 / 231 (11.00)	26 (100.00)
5 (16.90)	73 572.82 ± 444 227.02	34 / 445 (7.00)	25 (73.53)
3 (16.79)	73 444.51 ± 449 363.44	33 / 443 (7.00)	31 (93.94)

Table 5.12: Number of inclusion checks vs. speedup for *MinePump* false.

Setting	#*capsules*	# incl checks	speedup
Any	1	$8\ 624 \cdot 10^6$	1
No migr.	2	$24\ 460 \cdot 10^6$	0.54
M_1	2	$14\ 918 \cdot 10^6$	1.04
No migr.	3	$9\ 745 \cdot 10^6$	1.53
M_1	3	$19\ 115 \cdot 10^6$	1.08
No migr.	4	$16\ 056 \cdot 10^6$	1.09
M_1	4	$27\ 861 \cdot 10^6$	0.82

Table 5.13: Number of inclusion checks vs. speedup for *RCS5* false.

Setting	#*capsules*	# incl checks	speedup
Any	1	$263 \cdot 10^6$	1
No migr.	2	$223 \cdot 10^6$	2.05
M_4	2	$239 \cdot 10^6$	1.91
M_5	2	$197 \cdot 10^6$	2.21
No migr.	3	$337 \cdot 10^6$	1.97
M_4	3	$295 \cdot 10^6$	0.96
M_5	3	$296 \cdot 10^6$	2.24
No migr.	4	$280 \cdot 10^6$	2.72
M_4	4	$294 \cdot 10^6$	2.57
M_5	4	$252 \cdot 10^6$	2.97

5.4.1. Threads to Validity

The results presented here have been collected using only 4 processors, and do not involve any network at all. We don't believe that the test platform jeopardizes the validity of the results because:

- There is no claim of definitive positive outcome. If we would have claimed, for instance, that a particular method achieved linear speedups, then no conclusion could have been drawn until more tests were done on a different hardware platform, involving more processors.
- Interesting enough data could be collected an analyzed in the used platform. Although there is a (hypothetical) chance that with more processors some of the problems would not have presented themselves, they still need to be solved for the setting we used.
- The lack of a proper network does not seem to be such a pressing issue: TCP/IP is used anyway, so all the network stack overhead is payed. Also, if the packets were directed to another machine, at some point they would have been offloaded to the network card, freeing the main processor. In our setting they are always handled by the OS, and thus by a processor, which cannot return control to the application.

5.5. Conclusions and Future Work

Summing up, we can conclude that although migrations do not achieve linear speedups, and they are far from perfect, some kind of dynamic reconfiguration is indeed needed: Table 5.3 shows imbalance in the workload pattern of each *capsule*, Table 5.10 points out that some (few) locations are responsible for most of the work load and Table 5.11 indicates that they might not be the same along all the run.

Also interesting, the fragmentation phenomenon shows up again, although in a not so clear manner. As was already mentioned, more specific studies that collect particular metrics are needed to fully understand it. Nevertheless, for the forward algorithm, there are heuristics in the literature that seem to be able to counter it and we plan to explore them in the future.

We concluded Chapter 4 almost signaling the *end of road* for the distribution of the backwards algorithm. In this one, however, the landscape is completely different. It seems that we are only starting to understand the problems involved.

Previous section showed the wealth of variables that can be tuned to affect the verification speed. It also made it clear that there is no "winner setting", in the sense that different options benefit different case studies. We should be quick to point out that this is in no way privative of distributed approaches, and refer to Chapter 10 for a discussion on the subject.

Let's just advance here that a self-reconfiguring approach seems to make sense, specially considering that there are still not-so-big case studies that take many hours to complete.

One way to tune the $\#TVL$ variable would be to make it dynamic: keep analyzing visited states and add locations up to the point where the difference between the number of states per location (in successive locations) goes above certain threshold. For s_{TPL}, a binary search approach might make sense: if the prediction is not good, double it. If it is too time consuming, reduce it. Δt_m could be turned into a timer, making the *coordinator* an active component, instead of a passive one as it is today. Consequently, $\#s$ could be set to a high value and complemented by the *coordinator* requesting information when it needs it.

As witnessed by Table 5.10, heavy locations, accounting for most of the work, are in forward model checking as much of an issue as they were in backwards.

An appealing idea is the use of a hybrid approach, where a cluster of multicore processors (a very common architecture nowadays) runs a distributed verification, each node being multithreaded[6].

[6] A similar approach for the backwards setting is showing promising preliminary results.

Those multiple threads could explore states from *Pending* in parallel, or be used to parallelize the inclusion check in the *Visited* queue. This can be done by having each thread check against a different state from the queue, or by parallelizing the inclusion check itself. This last approach should not be discarded a priori, but the others seem to require less locking and easier contention avoidance.

Neither these approaches, nor the one that is presented next are as decoupled as some successful research lines in the untimed setting, where the data structure is completely partitioned among processing nodes in such a way that there is no need for them to coordinate results again (see, for instance, [HGGS02]). That does not seem possible with DBMs: even if each thread checks the inclusion of the new state against a different one from the *Visited* queue, they have to tell the others to stop checking as soon as the first check succeeds.

The purely distributed setting is nevertheless not exhausted. The work on Clock Reordering (see Section 8.1) seems to show that not all active clocks are created equal, and some of them have, for a given location, a broader range of possible values. This distinction could be the base for a technique that would allow to split a location among multiple processors. It would work on the following lines:

1. Predict clock ranges using the techniques of Section 8.1. This is a lightweight static analysis with very low resource impact.
2. Use GTPL and GTVL as suggested by Section 5.3.1 (or any other workload prediction mechanism that might be devised in the future) to find out the heaviest locations and the less loaded *capsules*.
3. Let l be a heavy location. Split it among the $c_1 \dots c_k$ *capsules* according to its clock range. For that purpose, consider the clock with the wider range for location l. Having a wider range, it is expected that its values be somewhat (hopefully, uniformly) distributed along the range. In other words, let's x_i be such a clock. If its predicted range is $[1, 200)$, it seems reasonable to expect that there will be more or less the same numbers of zones for location l with $x_i \in [1, 100)$ than with $x_i \in (100, 200)$. Caveat: the example should not be taken for a fact, as clock range prediction is more complicated. See Section 8.1 for an outlook.
4. When a new state (l, z) is discovered, send it to the appropriate *capsule* according to the value of the selected clock in z. Cases could arise when it should be sent to more than one *capsule*. Then that should be done, and some protocol should be put in place to deal with the situation.
5. Such protocol could be as simple as the following: although *capsules* $c_1 \dots c_k$ are in charge or checking whether (l, z) is new or not, only c_k is responsible for exploring it (this responsibility might be assigned in a round-robin fashion for each new state). So, if c_k finds that (l, z) is not new, it discards its. If it is new, it queues (l, z) for latter exploration: it still have to receive the agreement from $c_1 \dots c_{k-1}$. Those agreements could be batched, to reduce network usage.

The above mentioned ideas are purely speculative, mainly based on others that work separately. They show, however, that there is still work to do regarding the distribution of the forward algorithm.

In Section 6.1 we mention work by others that reached good speedups for the distribution of the same algorithm, and explain why we believe that they were not based on the multiprocessing itself, but on omitting inclusion checks. We also express our concern for a lack of a clear argument showing that termination is not affected by the approach, and wonder if similar speedups could be reproducible in a monoprocessor, just tuning the amount of omitted inclusion checks. Once these questions are settled, it will be worth exploring a distributed forward algorithm that combines both approaches.

Chapter 6

State of the art

This brief chapter surveys the state of the art in distributed model checking and compares our research with others'.

Although much successful work has been done to distribute *untimed* model checkers (see, for instance, [SD97, LS99, BBS01, GMS01, RSBSV96, BDHGS00, GHS01, HGGS02, BLW01, HKK02, Krc03], etc.), except for some work on a distributed version of UPPAAL [BHV00, Beh05] and our own [SBO02, BOS02, BOS05, BOS04a, BOS06b], not much has been previously done about parallelizing or distributing *timed* model checkers. A probable cause is that because of the inherently different data structures involved, the timed and untimed cases lead to distinct parallelization strategies and challenges.[1]

The idea of mixing distributed computing and model checking goes back to 1997 when Stern and Dill presented in [SD97] a distributed version of their (explicit state) Murφ model checker, achieving positive results. They used a hash function (the same universal hashing function that Murφ used to store states, actually) to map states to processors. The schema was straightforward: processors had a queue of states to be explored, took one from there, and sent the newly discovered ones according to the hashing function. Some buffering was included, and they managed to get very close to linear speedups[2]. Their work was so important that most distributed model checking algorithms can be thought of as a variation of it.

Since then, a lot of work has been done and the area is quite fertile, as can be witnessed by the continuity of the specialized "Parallel and Distributed Methods in verifiCation" series (see `http://pdmc.informatik.tu-muenchen.de/`), a yearly workshop that goes on since 2002.

It is worth mentioning a distributed version of the SPIN model checker [Hol03] that was presented in 1999 in [LS99] to deal with reachability and another in [BBS01] that handled LTL formulae. Their working was very similar to the distributed version of Murφ, with two notorious differences. They included a *manager* process, similar to our *coordinator*, to handle centralized decisions, and used a different hashing strategy: the hashing function was not the same as the one used for storage, it was only based on one of the component of the state vector.

Both articles presented scalability problems, but managed to verify previously untreatable case studies, because of the increased memory available. That early work showed that load balance was a key issue.

As was mentioned, the first attempts used hash functions on locations as a load balancing strategy. The problem we see with hash functions is that they don't provide enough control on the distribution,

[1] Although there is still work in monoprocessor timed automata model checking, and in distributed untimed model checking, we seem to be the only group who is actively working in the topic of distributed verification of timed systems nowadays.

[2] Remember that linear speedups means that if t_1 is time of a monoprocessor run, n processors take t_1/n.

being based only on statistical properties. This is exacerbated on symbolic-state model checkers because, contrary to explicit state ones, each symbolic-state can potentially have a different size or impose different workload (cft. [Beh05, BOS05], and Section 4.5.1).

In the untimed world, symbolic model checkers generally resort to BDDs as data structure. A distributed version of the SMV [BCM+92] model checker was presented in [BDHGS00, HGGS02] where a dynamic load balancing strategy –based on splitting BDDs– was used: when a node detects that it is very highly loaded, it finds an idle or almost idle one and "shares" his workload with it. Key to the success of this approach is the ability to split BDDs.

Unluckily, in the timed scenario the outlook is harder. Each symbolic state is composed by a location vector and a DBM. As was seen in Chapter 4, some algorithms (Algorithm 4.1.2 – region subtraction, for instance) are highly coupled thus making the "splitting" of a DBM or region a very complex task. Further, even the possible splitting is not as clean as in the untimed case, where once the BDD is partitioned, operations over it can be computed on only one node. With DBMs, most schemas would probably require processing to be done in many nodes and then have some sort of exchange to collect the results.

In spite of that, *some* research has been done about distributing timed model checkers. Year 2000 saw an article from Behrmann et al. [BHV00] in which a distributed version of UPPAAL [BLL+95] was presented. The paper dealt with forward reachability and used a hash function to balance load and the MPI library [For03] as a message passing mechanism to communicate between processing nodes. The article reported supralineal speedups (which were latter corrected, see below) but noted that the search order could impose an important difference on verification time.

Our own work pioneered the distribution of the backwards algorithm (see [Sch02, BOS02]). We managed to verify cases previously untreatable but found important obstacles to scalability, broadening the description of the search order issue previously described and introducing the phenomenon of *fragmentation* of the data structure (described in Section 4.3.2). An important difference is that we used an explicit distribution, which allowed us to try different load balancing strategies (see Section 4.2).

A latter article by Behrmann, [Beh02] (expanded latter in [Beh05]), attributed the supralineal speedups to the experiments on the distributed platform being "preliminar and inconclusive" (sic), described poor scalability in their initial implementation, confirming our discovery of the fragmentation phenomenon, and proposed a series of optimizations (including an explicit, probabilistic load balancer) leading to speedups between 50% to 90% of linear.

Although his results were promising, he only used four case studies with 1, 3 and 6 clocks (recall from Theorem 2.1 that the complexity is driven by the number of clocks). In the next section we present a more detailed comparison between this work and ours, and will discuss what we believe is the key aspect of their approach and how it relates to distribution.

6.1. Comparison with Other Tools

As was said previously there are not many other research works studying distributed *timed* model checking. Detailed comparisons against untimed tools do not make much sense: the data structures are so different that their handling should –most presumably– differ.

It is worth comparing our design decisions with the ones of the distributed version of UPPAAL ([Beh05]) by Behrmann, as both tools deal with distributed forward reachability over timed automata.

Behrmann's work also uses DBMs as data structures, works with a *Pending* and a *Visited* queue per processor, also unified, but stored as a hash table. He distributes locations to processors according to a hashing function (considering only the location vector). When a new state is discovered, it is sent to its owning processor, possible mediating some buffering. All the decisions are taken distributedly, as they lack a centralized monitor.

To get better balancing, they resort to a *proportional redirector* which, with some probability (involving load of the processors), sends a newly discovered state to some node other than its owner. The mentioned processor's load is actually the number of states in its *Pending* queue. The probability of a state being redirected to some processor other that its owner (call it p_1) increases with the difference between $load(p_1)$ and $load_{avg}$. A similar probabilistic analysis determines where the state is actually sent to. States received are not bounced back.

It is worth noting that when a state arrives to a processor that does not own it, it is *not checked* for inclusion in the *Visited* set. This is because such a set is only stored in the owning processor. These cases might lead to repeated exploration.

As there's no centralized decision maker, when processors exchange states they piggyback load information. This last decision raises the question of what happens if there's a highly loaded node, and one almost idle, but with no state exchange among them.

Our choice regarding distribution has been the use of an explicit mapping between locations and processing nodes. Although storing that mapping does indeed require memory proportional to the number of locations, it should be remembered that in timed model checking the control graph is many orders of magnitude smaller than the state space, so penalizing the *coordinator* with the centralized storage of that mapping imports an almost negligible strain on its resources.

Hashing states does have an advantage: it does not matter if a location is seen for the first time, it always has an owner. We, on the other hand, must collect it and then assign it (we explained that in detail in the proof of Theorem 5.1).

The choice made by Behrmann, although working fine in many cases, still has cases where it cannot achieve balance. Nevertheless, we believe the experiments conducted are not conclusive enough, because of the low number of clocks used. It might be the case that, had case studies with more clocks been used, thus increasing the number of zones per location, the results would have been worse. The rationale for that is not only that complexity is driven by the number of clocks, but also that more clocks increment DBM dimensions, and, potentially, the number of possible zone variation that can be discovered for a given location. In turn, it means more states redirected to processors that do not have the corresponding *Visited* set to check against, thus leading to more repeated exploration, which can jeopardize termination (see below).

The explicit mapping, on the other hand, allows us to apply direct load balancing measures, like the migrations mentioned in Section 5.3. We do pay an overhead by migrating the $Visited_l$ queue, but this same cost is considered by ParMETIS while deciding if the migration is worth doing. Anyway, from the evidence we collected in Section 5.4, the overhead seems very low. Also, we guarantee that no repeated explorations are performed.

Termination, if proper care is not payed to it, can be compromised if there is a load balancing algorithm that moves states. This, and not increased times, is the main risk of repeated exploration. Our load balancing algorithm does not have such a problem, and we proved in Theorem 5.2 that it does not jeopardize the correctness of the distributed exploration. A similar proof is lacking in Behrmann's article.

They seem to do a pretty good job on preserving locality – that is, that a state's successors be owned by the same processor, to reduce network transmissions. Being more concerned with load balancing, we haven't explored locality as a goal, what we speculate that we could deal with it, as suggested by the explicit mapping (for example, by informing ParMETIS of the real topology).

The UPPAAL team devotes a lot of work to prevent imbalances in its hash table implementation of the *Visited* queue. We, on the other hand, use an AVL tree [AVL62] sorted by location, for that purpose. Their worst case is proportional to the number of states, which –as was said before– is much greater than the number of locations. Ours, grows with the logarithm of the locations.

We mentioned in Section 5.2.3 how trace reconstruction is far from trivial in the distributed

Table 6.1: Percentage of successful inclusion checks

Example	% successful incl checks
RCS5 true	0.07
RCS5 false	0.04
FDDI4 true	46.75
FDDI4 false	0.77
FDDI8 true	30.95
FDDI8 false	2.94
Conv6ABC true	0.48
Conv6ABC false	0.05
Pipe6' true	0.40
Pipe6' false	2.05
MinePump true	0.01
MinePump false	0.01
MinePump' true	0.02
MinePump' false	0.01

environment. There is no mention of that aspect in Behrmann's work –maybe because all the case studies used were unreachable–, but finding where some father state lies at the end of the verification seems really hard using his proportional redirector, as there does not seem to be an easy way to tell where it is now, except asking to all of the processors.

The fact that some locations require significant more work than others was stressed repeatedly through this thesis. Behrmann's strategy does seem fit to deal with that, except for the cases where some few states are harder than the rest. According to Table 5.11 on Chapter 5 this is exactly the case, at least in all of our case studies. The effect in their architecture is that those states, when redirected, will not be checked against *Visited* (which reduces load), but will instead be explored again, generating more load. One scenario when this could have positive impact is where most of those inclusion checks were going to fail anyway. If that is the case, not doing them might actually be a good thing. Our evidence suggest that this is indeed the case (cf. Table 6.1).

The natural question to be asked is: is their speedup a result of distributed processing, or it comes from the fact that their proportional redirector is having the effect of avoiding inclusion checks that, statistically where going to fail anyway? The latter option seems really plausible. The more processors, more chance a state gets redirected, and thus avoids to be checked, going directly to be explored.

An interesting sequel: what happens if, in a monoprocessor setting, some inclusion checks are randomly ignored, assuming they fail? Will the same phenomenon manifest? We consider this a very interesting research question.

Chapter 7

rCDDs

In this chapter we introduce *rCDDs*, a variation of the Clock Difference Diagrams presented in Section 3.4, as a data structure for model checking timed automata. We present and prove correct the complete set of operations required to do forward reachability analysis without the need for DBMs, which are traditionally required. By employing a fully rCDD-based algorithm our experiments show a reduction of around 40% in time requirements (with only a moderate increase in memory consumption) in current case studies from the literature.

Although representing non-convex sets being the original motivation for rCDDs, they present some difficulties, which we explore, also proposing alternative solutions.

The work on this chapter was Esteban Pavese's degree thesis [Pav06] (see Section 1.9). The actual chapter is based on an article still being prepared.

7.1. Introduction and Previous Work

As was said before, for both warranting termination and avoiding duplicated exploration, timed automata reachability requires knowing if a newly discovered symbolic state is already covered by the existing ones. This is challenging for DBMs, because they can only represent convex clock valuations. The traditional solution is to employ sets of DBMs, but it is usually very hard to detect superpositions there, leading to repetitions of calculus and other inefficiencies.

There have been many attempts to overcome this problem by using BDD-like data structure, which we summarized in Section 3.4. The most successful, was a 1999 article by Behrmann et al. In [BLP+99] they presented *Clock Difference Diagrams* (CDDs for short). Their work used DBMs for most of the operations and CDDs for the *Visited* set, obtaining memory savings at the cost of extra time.

We consider that CDDs needed to be revisited because:

- An algorithm that did not rely on DBMs was yet to be provided.
- Correctness proofs were lacking for the new data structure.
- The original case studies used only systems with a maximum of 5 clocks. Complexity is $O(n!2^nC^n)$, where n is number of clocks and C is the largest constant appearing in the inequalities (cf. Theorem 2.1).
- Being a tree-like data structure, they could have repetead subtrees. CDDs require that repeated subtrees were replaced by aliasing in a maximal way (refered to as "maximal sharing" in the original article). However, detecting repeated subtrees can be expensive, and we believe that the claim of being able to do so by a simple constant time lookup in a hash table is insuficient and needs further clarification.

In this work, we present a data structure called *Relaxed CDD* (*rCDD* for short) in which we

implement all the operations required for the above mentioned reachability algorithm. rCDDs are similar to CDDs, but do not require maximum sharing of subtrees. They achieve the reduction of verification times at the expense of storing some extra constraints (compared to the traditional encoding of DBMs). Their usage is explored in two ways: as a representation for both convex and non-convex sets of clock valuations. We present correctness proofs and run case studies from the current literature obtaining speedups of up to 40% with moderate increase of memory overhead.

CRDs, short for *Clock Restriction Diagrams*, should also be mentioned. They were introduced by Wang in [Wan03b] and are similar to CDDs, but use open intervals. As will be shown in Section 7.2.5, CDDs allow to avoid many calls to a cubic function, which is key to the time savings.

7.2. rCDDs

In this section, after presenting the basic definitions of our data structure, some methods to reduce the traditional memory overhead of tree-like structures are shown in Section 7.2.2. They are specialized for zones in Section 7.2.3. Section 7.2.4 covers the algorithms and their correctness. Special focus is done in Section 7.2.5 on how to avoid some costly operations in rCDDs, which is key to their performance. Finally, Section 7.2.6 contains a discussion on non-convex set representation.

7.2.1. Data Structure Definition

Definition 7.1 (rCDD)
Given a set X of clock variables, $X' = X \cup \{0\}$, an rCDD is defined by a tuple $k = [diff(k), Ints(k), S(k)]$, where:

1. *$diff(k) \in (X \times X') \cup \{TRUE, FALSE\}$ is the clock difference represented by this node. These clock differences are extended with values TRUE and FALSE. We call nodes that hold the latter* terminal.
2. *$Ints(k)$ is a list of outgoing edges. Each edge is labeled with an integral interval. $Ints(k)_n$ denotes the nth list's interval, and $\#_{Ints}(k)$ denotes the list's size.*
3. *$S(k)$ is a list of successor nodes. $S(k)_n$ and $\#_S(k)$ are defined in a similar way as previously. For each $1 \leq n \leq \#_S(k)$, $S(k)_n$ yields the node reached by traversing the edge $Ints(k)_n$.*

An rCDD k such that $diff(k)$ is *TRUE* (or *FALSE*) will usually be referred as just *TRUE* (or *FALSE*) for short. Similarly, given an interval I we will write $S(k, I)$ to denote the node obtained by traversing the edge labeled I from k. Formally, $S(k, I) = k' \Leftrightarrow \exists i, 1 \leq i \leq \#_{Ints}(k)$ such that $Ints(k)_i = I \wedge S(k)_i = k'$.

Since rCDDs are hierarchical in nature, an order on clock differences is needed, which can be defined as an extension of an order on clock variables.

Definition 7.2 (Total Order on Clock Differences)
Given a total order $<$ on clock variables, the total order on clock differences, also noted $<$, is defined as its pairwise extension.

This total order allows us to state the structure's invariant, as follows:

Definition 7.3 (Structure Invariant)
The following constraints are imposed on every node k of an rCDD:

- *Whenever $diff(k)$ is TRUE or FALSE, the lists $Ints(k)$ and $S(k)$ must be empty.*

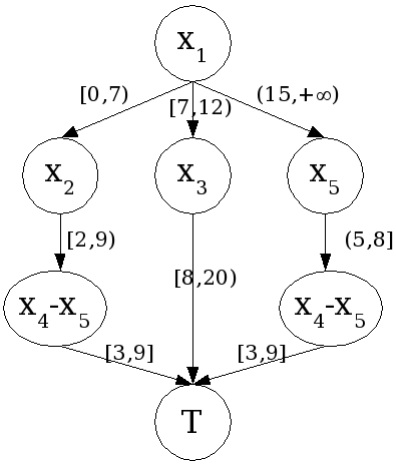


Figure 7.1: A (non-convex) rCDD for the region $(0 \leq x_1 < 7 \wedge 2 \leq x_2 < 9 \wedge 3 \leq x_4 - x_5 \leq 9) \vee (7 \leq x_1 < 12 \wedge 8 \leq x_3 < 20) \vee (15 < x_1 < \infty \wedge 5 < x_5 \leq 8 \wedge 3 \leq x_4 - x_5 \leq 9)$.

- *In the other case, no element of $S(k)$ may be FALSE.*
- *Given a total order on clock differences $<$, it holds that for any k' in $S(k)$, either $diff(k) < diff(k')$ or k' is TRUE.*
- *Since edges are labeled by intervals, the existence of both differences $x_i - x_j$ and $x_j - x_i$ for any pair of clocks x_i, x_j is redundant (because each one can be inferred from the other). Therefore, if $diff(k)$ is of the form $x_j - x_i$ it must hold that $x_i < x_j$, to guarantee uniquness.*
- *Every pair of intervals in $diff(k)$ must be disjoint. Moreover, $diff(k)$ must be sorted; we say that for any two intervals I and J, $I < J \Leftrightarrow \forall i, j$ such that $i \in I$ and $j \in J$, it holds that $i < j$.*

It is worth noting that, unlike CDDs, rCDDs do *not* require sharing of repeated subtrees. Fig. 7.1 shows an example rCDD. On the following, we will elaborate on the structure's semantics. For this purpose, we'll need to define a mapping between temporal constraints in Ψ'_X and rCDDs.

Definition 7.4 (Mapping of Temporal Constraints to rCDDs)
$\forall \psi_1, \psi_2 \in \Psi'_X, x, y \in X, x < y, c \in \mathbb{N}$, we define the mapping $r \rightsquigarrow rCDD$ as follows:

$$r \rightsquigarrow rCDD(True) = [TRUE, \lambda, \lambda] \tag{7.1}$$
$$r \rightsquigarrow rCDD(False) = [FALSE, \lambda, \lambda] \tag{7.2}$$
$$r \rightsquigarrow rCDD(c < x) = [x - 0, \langle\langle (c, \infty) \rangle\rangle, \langle\langle r \rightsquigarrow rCDD(True) \rangle\rangle] \tag{7.3}$$
$$r \rightsquigarrow rCDD(x < c) = [x - 0, \langle\langle [0, c) \rangle\rangle, \langle\langle r \rightsquigarrow rCDD(True) \rangle\rangle] \tag{7.4}$$
$$r \rightsquigarrow rCDD(x - y < c) = [x - y, \langle\langle (-\infty, c) \rangle\rangle, \langle\langle r \rightsquigarrow rCDD(True) \rangle\rangle] \tag{7.5}$$
$$r \rightsquigarrow rCDD(\psi_1 \wedge \psi_2) = Intersection(r \rightsquigarrow rCDD(\psi_1), r \rightsquigarrow rCDD(\psi_2)) \tag{7.6}$$

Definitions 7.3, 7.4 and 7.6 have also a corresponding one for $\leq$ with the interval closed to the right.

We will now define the semantics of the structure. This task will be rendered easier by providing some notions on rCDD traversal.

Definition 7.5 (rCDD Path)
A path *on an rCDD k is a sequence $P[1] \ldots P[n]$, where each element is a tuple $\langle clocks, interval \rangle$, such that:*

- *$clocks(P[1])$ is either a clock difference, TRUE, or FALSE.*

- *interval*$(P[1])$ *is an interval* $\subseteq \mathbb{R}$.
- *Either diff*$(k) =$ *clocks*$(P[1]) \in \{TRUE, FALSE\}$ *and length*$(P) = 1$*; or all the following hold:*

  1. *diff*$(k) =$ *clocks*$(P[1]) = x_j - x_i$ *for some* x_i, x_j*; and*
  2. *interval*$(P[1]) \in$ *Ints*(k)*; and*
  3. $P[2] \ldots P[n]$ *is a path in* $S(k, interval(P[1]))$.

Definition 7.6 (rCDD Path Satisfiability)
Given an rCDD k*, a path* P *in* k *and a clock valuation* $v \in \mathcal{V}_X$, v satisfies P *(noted* $v \models P$*) if and only if for each* i, $1 \leq i \leq length(P)$ *either* $clocks(P[i]) = TRUE$*, or else* $clocks(P[i]) \notin \{TRUE, FALSE\}$ *and* $v(clocks(P[i])) \in interval(P[i])$.

Definition 7.7 (rCDD Satisfiability)
The satisfiability of an rCDD k *by a clock valuation* $v \in \mathcal{V}_X$ *(noted* $v \models k$*) is defined inductively as follows,* $\forall x, y \in X, c \in \mathbb{N}, d \in \mathbb{Z}$:

$$v \models [TRUE, \lambda, \lambda] \quad \wedge \quad v \not\models [FALSE, \lambda, \lambda] \tag{7.7}$$

$$v \models [x - y, Is, Ss] \quad \equiv \quad \bigvee_{i=1}^{\#(Is)} \{ \, (v(x) - v(y) \in Is_i \,) \wedge v \models Ss_i \, \} \tag{7.8}$$

Observation 7.1 *It is worth emphasizing that given the intervals' disjunction, the satisfied path (if any) is unique.*

Theorem 7.1 (Correctness)
The mapping $r \rightsquigarrow rCDD$ *previously defined is correct with respect to satisfiability, that is,* $\forall \psi \in \Psi_X, v \in \mathcal{V}_X, v \models \psi \Leftrightarrow v \models r \rightsquigarrow rCDD(\psi)$.

Proof

The proofs can be found in [Pav06].

□

7.2.2. Reducing Memory Consumption

rCDDs are basically trees. Tree-like structures can have a large representation overhead if they are not treated carefully.

Observation 7.2 *Each rCDD node (k) can be represented in a fixed number of bits. As the number of clocks is fixed it is easy to compute how many bits are needed to represent two clocks.*

■

Observation 7.3 *Each rCDD edge* $(Ints(k)_i)$ *can also be represented by a fixed number of bits. Each comparison requires an additional bit to indicate whether the constraint is* $<$ *or* $\leq$*. To calculate how many bits will be needed for the constants, use the upper bound as defined in [BBLP04]. After that, a pointer to the* $S(k)_i$ *node is needed. Pointers have a fixed length (given an architecture).*

■

Because of Observation 7.2 and Observation 7.3, we are able to code each node and its lists of successors in a compact way, very similar to what we did for RDBMs in Section 3.3. We will see in Section 7.2.3 how the memory requirements can be further reduced for zones.

```
1: function UNION (r_A, r_B : rCDD)→ r' : rCDD
2:     if r_A = FALSE then return rCDD_B
3:     else if r_A = TRUE then return TRUE
4:     else if diff(r_A) < diff(r_B) then
5:         for all I ∈ Ints(r_A) do
6:             Add_rCDD(r', I, UNION(S(r_A, I), r_B))
7:         end for
8:     else if diff(r_A) > diff(r_B) then
9:         ...follows as previous case.
10:    else
11:        for all I ∈ Ints(r_A), J ∈ Ints(r_B) do
12:            if ¬Empty(I ∩ J) then
13:                Add_rCDD(r', I ∩ J, UNION(S(r_A, I), S(r_B, J)))
14:            end if
15:        end for
16:    end if
17:    return r'
18: end function
```

Algorithm 7.2.1: rCDD union recursive algorithm.

7.2.3. Representing Zones

Zones have a very particular shape in rCDDs: each node has only one edge. We can take advantage of that to produce a more compact representation. As we only need to code for a list of nodes (pair of clocks) and intervals, there is no need to specify pointers: each interval refers to the next node on the list.

From Observation 7.2 and Observation 7.3 we know that each pair (node, interval) can be coded in a fixed number of bits. So, we can calculate a priori how many intervals can be packed in a machine word. If we also consider that the minimized version of the constraint system usually requires $O(n)$ clock constraints, we can allocate continuous blocks of words so that most rCDDs will fit in just one of them. If more than one block is needed, only then a pointer is required.

In this way, not only memory is saved (in modern machines a pointer takes a considerable number of bits –e.g. 32 or 64– which can instead probably code many intervals), but also less pointers are dereferenced, making the algorithm, at the assembler level, operate more time on registers and less on memory, thus also saving time.

In the remainder of this chapter an rCDD representing a zone will be noted as rCDD_z.

7.2.4. Algorithms

In this section, we present in Algorithms 7.2.1 to 7.2.4 the conceptual, recursive algorithms for *all* operations needed for reachability verification with rCDDs (except for inclusion, which is presented in its iterative version in Algorithm 7.2.5). In these algorithms, the restriction forbidding the existence of *FALSE* nodes has been relaxed, in order to keep the presentation simple. We also elaborate on the strategies used to develop efficient, iterative algorithms, where the restrictions are enforced in full. Finally we present the iterative algorithm for zone inclusion checking, being a critical operation in the verification, along with a proof of its correctness.

Algorithm 7.2.1 (union) decides which clock difference is first according to order given by $<$. Suppose its the one on r_A (line 4). Then, in the resulting rCDD, each interval in r_A has to lead to the union of r_B and the corresponding sub-rCDD in r_A. The treatment is similar if r_B is the first (line 8).

```
1: function RESET (C : Set(clock), z : rCDD_z) → z' : rCDD_z
2:     if z = FALSE ∨ z = TRUE then return z
3:     end if
4:     I ← Ints(z)_1
5:     rCDD_z z'' ← RESET(C, S(z, I))
6:     if diff(z) = x − 0 for some x ∈ C then
7:         diff(z') ← diff(z)
8:         AddInterval (z', [0, 0], z'')
9:     else if diff(z) = x − y and (x ∈ C ∨ y ∈ C) then
10:        if y ∈ C then
11:            diff(z') ← diff(z)
12:            AddInterval (z', CLOCKINTERVAL (x, z), z'')
13:        else
14:            diff(z') ← (−1)×diff(z)
15:            AddInterval (z', CLOCKINTERVAL (y, z), z'')
16:        end if
17:    else
18:        AddInterval (z', I, z'')
19:    end if
20:    return z'
21: end function
```

Algorithm 7.2.2: rCDD clock reset recursive algorithm.

If they are the same (line 10), an interval needs to be added to the result for each intersecting pair of intervals in r_A and r_B. The intersection (shown in Algorithm 7.2.3) follows the same pattern.

Algorithm 7.2.2 (reset) is a straightforward recursive implementation of the reset successors transformation explained in Section 3.1. As it traverses the path (it operates on zones, which have only one), it looks for intervals involving the clocks being resetted. The ones where the other clock is 0 (line 6) have their interval replaced by $[0, 0]$ in the resulting rCDD (line 8). If the other clock is not clock 0, the interval is replaced in the result by the one corresponding to the other clock (the one that is not resetted – see line 12 and line 15).

As Algorithm 7.2.4 (time successors) operates on zones, it traverses the only path in the rCDD looking for clock differences involving the clock 0 (line 3). Each of them has only one interval, which is copied to the resulting rCDD with its upper bound turned into ∞ (lines 5 to 8).

An explanation of Algorithm 7.2.5 (inclusion checking) is delayed until Theorem 7.3, where it is presented along with its correctness proof.

Theorem 7.2 (Algorithms' Correctness)
The algorithms presented in Algorithms 7.2.1 to 7.2.4 are correct. That is, the only valuations that satisfy the resulting rCDDs are the ones that are compatible to the application of the respective operation in the input constraint system.

Proof

The proofs can be found in [Pav06].

□

Iterative versions of the algorithms are stack-based. The stack's size is fixed during the verification process, thus reducing memory operations on it (the maximum stack size is roughly $n^2/2 + n$ where n is the number of clocks in the system, though it will rarely reach that size). Also, only intervals taking part on non-*FALSE* paths are stored and evaluated, effectively reducing memory and time

```
function INTERSECTION (r_A, r_B : rCDD)→ r' : rCDD
    if r_A = FALSE then return FALSE
    else if r_A = TRUE then return r_B
    else if diff(r_A) < diff(r_B) then
        for all I ∈ Ints(r_A) do
            Add_rCDD(r', I, INTERSECTION(S(r_A, I), r_B))
        end for
    else if diff(r_A) > diff(r_B) then
        ...follows as previous case.
    else
        for all I ∈ Ints(r_A), J ∈ Ints(r_B) do
            if ¬Empty(I ∩ J) then
                Add_rCDD(r', I ∩ J, INTERSECTION(S(r_A, I), S(r_B, J)))
            end if
        end for
    end if
    return r'
end function
```

Algorithm 7.2.3: rCDD intersection recursive algorithm.

```
function TIMESUCC (z: rCDD_z)→ z' : rCDD_z
    if z = FALSE ∨ z = TRUE then return z
    else if diff(z) = x − 0 for some x then
        I ← Ints(z)_1
        J ← ⟨I.min, +∞⟩
        diff(z') ← diff(z)
        AddInterval (z', J)
        S(z', J) ← TIMESUCC(S(z, I))
    else return z
    end if
    return z'
end function
```

Algorithm 7.2.4: rCDD time successors recursive algorithm.

consumption. Algorithm 7.2.5 is used to decide whether a zone is included in a given region or not. The stack used in this case holds information about both rCDDs, marking intervals as they are traversed. The algorithm requires the rCDD representing the zone to be in canonical form, which is much like the canonical form for DBMs; this is not needed for the rCDD representing the region.

Theorem 7.3 (Zone Inclusion Checking Correctness)
The zone inclusion checking algorithm (7.2.5) is correct.

Proof

First of all, the algorithm termination is guaranteed since on each step smaller rCDDs are pushed onto the stack (or else they're pushed with more marks on its intervals). This ensures that eventually the stack will empty. We need to prove then that the result is correct, that is, $\forall \psi_1, \psi_2 \in \Psi'_X, \textsc{Included?}(Canonical(r \rightsquigarrow rCDD(\psi_1)), r \rightsquigarrow rCDD(\psi_2)) \Leftrightarrow \psi_1 \subseteq \psi_2$. The canonical form for an rCDD representing a zone is similar to that of a DBM, and the algorithm is essentially the same than that of $cf()$. We'll prove each implication.
$\Longrightarrow$) Assuming the algorithm returns *True*, we prove that the rCDD representing a zone has its only

```
1: function INCLUDED? (z: rCDD_z, r: rCDD)→ ret: boolean
2:     ret← True
3:     Stack(rCDD, rCDD) stack
4:     Push (stack, [z, r])
5:     while ¬ Empty(stack) ∧ ret do
6:         top ← Pop (stack)
7:         if top.r = FALSE then ret ← (top.z = FALSE)
8:         else if top.r = TRUE then ret ← True
9:         else
10:            if diff(top.r) < diff(top.z) then ret ← False
11:            else if diff(top.z) < diff(top.r) then
12:                I ← Ints(top.z)_1
13:                Push (stack, [S(top.z, I), top.r])
14:            else
15:                I ← Ints(top.z)_1
16:                if ¬ ISSPANNED? (I, top.r) then ret ← False
17:                else
18:                    J ← first unmarked interval in top.r
19:                    if there is no such J then continue
20:                    else
21:                        Mark (top.r, J)
22:                    end if
23:                    if I ∩ J ≠ ∅ then
24:                        Push (stack, [top.z, top.r])
25:                        Push (stack, [S(top.z, I), S(top.r, J)])
26:                    else
27:                        Push (stack, [top.z, top.r])
28:                    end if
29:                end if
30:            end if
31:        end if
32:    end while
33:    return ret
34: end function
```

Algorithm 7.2.5: rCDD inclusion checking iterative algorithm.

path included (possibly split up) in those in the rCDD passed as a region. We'll analyze each case as we pop from the stack, beginning with the initial state of the stack, which has only $[z, r]$.

- if r is either *TRUE* or *FALSE*, the inclusion verification is trivial.
- if $\mathit{diff}(top.r) < \mathit{diff}(top.z)$, since $top.z$ is in canonical form this means the difference in r is restricted to some values, whereas z does not restrict it at all. This would make the algorithm return *False*, which goes against our assumption.
- if $\mathit{diff}(r) > \mathit{diff}(z)$, then the difference in z is not directly restricted in r (it may be, as deduced from other restrictions; however, we may ignore this condition until we find the conflicting differences). In this case we push the successor node for z.
- finally, if $\mathit{diff}(r) = \mathit{diff}(z)$ we only need to check that the interval leading out of z is spanned by those leading out of r. If that is not the case, then valuations exist that satisfy z but not r, which would have resulted in a *False* output. In order to check this, we push into the stack each successor of r that comes from an interval which spans over the one of z.

Since all options which would allow valuations satisfying z and not r are filtered out by the assumption of a *True* output, we only need to prove the other implication.

$\Longleftarrow$) We prove this implication by means of its counterpart. Therefore, we'll prove $\forall \psi_1, \psi_2 \in \Psi'_X, \neg\textsc{Included?}(Canonical(r \rightsquigarrow rCDD(\psi_1)), r \rightsquigarrow rCDD(\psi_2)) \implies \psi_1 \nsubseteq \psi_2$, that is, there exists a valuation v that satisfies ψ_1 but *not* ψ_2. Since the *ret* variable is set to *True* at the beginning, it will suffice to analyze the situations where it is set to *False*:

- it may be that $top.r = FALSE$ and $top.z \neq FALSE$. In that case any valuation v is such that it does not satisfy *FALSE*.
- it may also be that $diff(top.r) < diff(top.z)$. In that case, any valuation will satisfy $diff(top.r)$ in $top.z$: its absence means the lack of restriction. We may choose any valuation v such that $v(diff(top.r))$ is not in any of $top.r$'s intervals (there exist infinitely many of such valuations).
- lastly, it may be that $diff(top.r) = diff(top.z)$ and that the interval leading out of $top.z$ is not spanned by those in $top.r$. In this case, we just need to take a valuation v such that $v(diff(top.r))$ is indeed within the subinterval not spanned.

Since we find a valuation v for every case where the result is set to *False*, the theorem is proven.

□

7.2.5. Early Cut

The inclusion algorithm features early cuts in cases where it can be easily deduced that there is no inclusion. This is not news, as its DBM counterpart does the same. What's novel, is the use of an early cut in the intersection algorithm.

At the end of the intersection it is necessary to determine if the result is empty or not, and this requires obtaining the structure's canonical form. As was said in Section 3.1, $cf()$ is very expensive: its runtime is $O(n^3)$, n being the number of clocks in the system.

Because both upper and lower explicit bounds are present for each clock in rCDDs, it can be determined early in the process whether the resulting intersection will be empty or not. For instance, if $2 < x \wedge x < 3 \wedge y < 7$ is to be intersected with $0 < x \wedge x < 2 \wedge y < 3$, in rCDDs the upper and lower bound on x are close to each other in the representation, and thus operated at the same time, so only by performing the simple intersection operation, which means taking the stricter constraint for each bound, it can be deduced $2 < x \wedge x < 2$. Knowing that, the costly canonical form function has no need to be called.

The actual implementation of this early cut would be, in the iterative version of Algorithm 7.2.3, to return *FALSE* as an `else` to the `if` in line 12.

These savings in the number of times that the canonical form operation is called are key in reducing the verification time.

It should be noted that the same optimizations could in principle be applied to raw DBMs. However, they are incompatible with the minimal constraint representation. Having the intervals, rCDDs achieve not to store the complete matrix, but at the same time allowing for this important saving in computation.

7.2.6. Representing Regions

Although the main motivation of CDDs was to obtain an appropriate data structure for non-convex sets, some problems require special attention. We find the requirement for maximum sharing –present in CDDs and omitted in rCDDs– problematic.

Example	Components	Clocks	Reachable states
MinePump	10	10	1172911
MinePump'	10	10	131261
RCS5	8	8	271771
Pipe6	14	14	80280
Pipe6'	14	14	24375

Table 7.1: Examples sizes.

Example	RDBM based			With rCDDs		
	Time (secs)	Mem (MB)	$\#cf()$ ($\times 10^3$)	Time (secs)	Mem (MB)	$\Delta\ \#cf()$ (%)
RCS5 true	52.40	7320	273	56.49 (+7%)	9548 (+30%)	-38
RCS5 false	1859.40	34348	3784	1575.20 (-15%)	38144 (+11%)	-33
MinePump' true	71.71	6924	405	40.76 (-43%)	8112 (+17%)	-35
MinePump' false	856.84	19488	1728	554.91 (-35%)	23856 (+22%)	-33
MinePump true	5145.56	56644	5031	2977.80 (-42%)	72760 (+28%)	-33
MinePump false	37768.39	152840	15583	22440.00 (-40%)	194104 (+27%)	-32
Pipe6' true	128.66	16988	673	120.34 (-6%)	24560 (+44%)	-34
Pipe6' false	58.87	13228	349	40.81 (-30%)	11772 (-11%)	-39
Pipe6 true	4215.56	152416	10532	3366.45 (-20%)	200256 (+31%)	-40
Pipe6 false	253.29	35268	1168	160.91 (-36%)	30368 (-13%)	-42

Table 7.2: Results obtained with RDBMs vs. rCDDs.

Case studies have grown since the CDD article by Behrmann et al.. Nowadays tools need to routinely deal with models that have at least an order of magnitude more states, and many times the number of clocks (remember from Theorem 2.1 that timed automata reachability is exponential in the number of clocks).

Also, more operations output rCDDs –as opposed to only union in the CDD article–, meaning that repeated subtrees need to be detected on a number of different algorithms. Combined with the size of current models, we find that repetition detection consumes considerable time (it should be noted that [BLP+99] does not present any details on an efficient technique for finding them).

However, we still believe that sharing is *the* crucial point for the use of CDD-like data structures as a representation for regions. Our tests confirm that the number of inclusions and states is reduced using rCDDs as a representation for regions. However, memory requirements increase unacceptably. We implemented prototypical (i.e., proof-of-concept) repetition detection techniques, but they took too much time. We find that more research should be done in this area before reaching to a sustainable conclusion.

7.3. Case Studies

To validate the data structure, we incorporated rCDDs into a monoprocessor version of our model checker Zeus and ran a series of experiments against well known case studies from the literature, detailed in Appendix A.

Table 7.1 summarizes the sizes of the examples used in this chapter. All the experiments were run on an Intel Pentium IV 3.0 MHz machine with 2 GB of RAM, running Linux 2.6.

Table 7.2 presents a comparison against the standard version (RDBM-based) and one that uses rCDDs. The columns show total time, memory, and number of calls to $cf()$ in the RDBMs-based

version, and total time, memory and percentage of difference in the number of calls to $cf()$, for the rCDD-based version.

As can be seen in the table, in most of the case studies a time saving of around 40% is achieved, with the exception of a few cases that take little time and where overhead probably dominates. The same table also shows that the time saved is proportional to the number of avoided calls to $cf()$ (recall Section 7.2.5). There is, however, a memory penalty of more or less the same order of magnitude than the time saved. It is nevertheless interesting to note that in cases like *MinePump* false, the time saved is more than 4 hours of almost 11 hours, at the price of less than 40 extra MB of RAM, a tradeoff that seems more than acceptable.

For a number of technical and availability reasons we didn't have a chance to compare against other tools. However, as each tool contains heuristics and improvements that might not be present in the others, cross tool comparisons does not permit to assess the performance of the data structure by itself, which is the objective of this study. It should be noted, however, that the use of rCDDs is orthogonal to most available techniques for the reduction of time and space such as zone widening, abstractions, etc.

7.4. Conclusions and Future Work

In this chapter we revisited Clock Difference Diagrams, a data structure that was born to represent non-convex sets of clock valuations for model checking of timed automata. The original presentation left some questions open, namely the provision of an algorithm that did not rely on DBMs, experimentation with bigger case studies, and clarification of the repeated subtree detection algorithm.

While analyzing these questions we developed rCDDs, a variation of the above mentioned CDDs. rCDDs differ from standard CDDs, mainly, in that they do not require maximum sharing. Although that is a desirable property, repeated subtrees detection can be very costly. As all the algorithms needed for reachability analysis of timed automata have been presented and proven correct, rCDDs can completely replace the traditional DBMs as a data structure for timed model checking.

The original 1999 article that introduced CDDs, by Behrmann et al., showed memory savings at the price of augmented times. Their case studies, although representative at the time, are orders of magnitude smaller that the ones that tools deal with today. With hundreds of thousands of states, and many operations manipulating rCDDs, detecting repetitions becomes prohibitive. In fact, early tests confirm that the number of inclusions and states is reduced using rCDDs as a representation for non-convex sets, but, without sharing repeated subtrees, memory requirements increase unacceptably. However, we were not able to come up with a repetition detection technique that did not took also unacceptable extra time. It is worth noting that the original presentation of CDDs did not give details on how to achieve that. We still consider this an open question where more research should be put.

Special care has been put in the representation of zones, so that early detection of empty zones while computing the intersection is possible, avoiding costly canonicalizations, and thus saving time. Also, as zones have no ramifications, a very compact representation is used, which not only compensates for the extra constraints stored, but produces time saving on its own.

Case studies for rCDDs have shown a positive impact: times are reduced with a mild memory overhead. The savings go up to 40%. For instance, in cases like *MinePump* false, the time saved is more than 4 hours of almost 11 hours, at the price of less than 40 extra MB of RAM, a tradeoff that seems more than acceptable.

Future research agenda includes looking more closely into sharing for very large data structures. Partial hashing of subtrees might be used to detect duplication. Another idea is to store not the subtree, but the symbolic calculation from where it can be reconstructed. Also, as Zeus was born with the aim of being a distributed tool, we would like to look into the possibility of having multiple

threads working with different branches of the tree in parallel.

Chapter 8

Other Optimizations

This chapter browses through some model checking optimizations that were researched while working on the distributed forward algorithm described in Chapter 5. The good news is, they are also valid for the monoprocessor case.

An essential operation in timed automata model checking is *inclusion checking* which decides whether a set of states, represented as a convex polyhedron, is included in another set. As was already mentioned, several verification tools implement convex polyhedra as DBMs.

Inclusion checking can be called hundreds of millions of times during the verification of a medium-size model. The naïve implementation scans each matrix cell by cell and compares it against the corresponding one in the other matrix. If all the checks are successful the first matrix is included into the second. If one of them fails, it is not. In the last case, the order in which matrices are traversed is decisive for the inclusion checking's efficiency.

Next section deals with *clock reordering*, a technique published in [BOS06c] that reduces the number of comparisons needed to find a failure. Experiments show negligible memory overhead and time savings of up to 17%.

The technique was depicted while trying to deal with fair workload distribution in the distributed setting. As mentioned in Observation 4.2 and Table 5.10, there are locations which require many orders of magnitude more work than others (we called them *heavy*), so it seems that there is no hope in achieving balance if this work cannot be divided. How can a location be dealt with by many *capsules*? The idea was like this: suppose location l is one of those heavy ones, and we want to split it among *capsules* c and c'. To achieve balance, let's pick a clock x, whose range is, say, $[0, 200)$ and assign the zones that have $x \in [0, 100)$ to c and the ones with $x \in (100, 200)$ to c'.

Many challenges arise, but an interesting one is to find a clock x whose values are spreaded enough into the interval so all the work does not land in one *capsule* anyway. A good candidate clock should not only be active (in the sense of [DY96]), but should ideally be the *most active*. Looking at this idea turned into the work of next section. As the reader might have deduced, this work is a consequence of pursuing some of the ideas mentioned as future work in Section 5.5.

After that, Section 8.2 presents hypervolume approximation –published in [BLOS07]– which actually leads into two techniques: One of them is very simple to implement. The other, an improvement over the first, requires more involved programming. Each of them saves verification time (up to 19% in our case studies), with a modest increase in memory requirements. Their impact differs among the different case studies but, as they can be combined, there is no need to choose a priori.

8.1. Clock Reordering

8.1.1. Introduction

For both warranting termination and avoiding duplicated exploration, timed automata reachability (for instance, Algorithm 5.1.1) requires knowing if a newly discovered symbolic state is already covered by the existing ones. To determine this, an *inclusion check* is performed when a matrix is added to the set. The original algorithm for inclusion checking of DBMs scans each cell of the newly discovered matrix and compares it against the corresponding one in each existing DBM, (one at a time, see Algorithm 5.1.2), determining that the inclusion holds if every check succeeds, or does not hold when the first check fails. In the last case, the order in which the comparisons are done is very important, and that is why we would like to increase the chances of finding the failure as soon as possible by giving an appropriate ordering for cells. The inclusion operation's speed is critical for reachability, as it can be called hundreds of millions of times for medium-size models. As Zeus represents zones as linked lists of bounds (recall Section 3.2), the actual reorder is easy to implement, and the inclusion checking algorithm does not need to be modified.

8.1.2. Clocks Ranges as Predictors

By a simple probabilistic reasoning it can be seen that if every value in the clock's range had the same probability, the higher the range, the bigger the chance a bound involving this clock would fail in the inclusion check.

However, two problems arise: computing the actual range of clock values can be as hard as solving the reachability problem[1] and in many cases probability is not uniformly distributed in the range (for instance, it is very easy to construct an automata where a clock takes only even values).

Nevertheless, our technique tries to compute the range as tight as possible resorting only to local information (i.e., without considering the interaction with the other automata that comprise the system). Although the risk of a non-uniformity is still present, it seems that in practice it is not such a pressing issue. This is one of the key points that needs to be investigated further.

The aim is to *weight* clocks in a way such that higher weights means more probability of being a clock that makes the inclusion check fail.

8.1.3. Weighting Clocks

It is necessary to establish the weight of each clock, for each location and each transition. Because each timed automata in the system under analysis has a separate set of clocks, the weight can be computed independently, and then combined when the on-the-fly composition takes place (see bellow in this section).

It should be noted that in timed model checking, even when the composed system has millions of symbolic states, each component generally has a small control graph with rarely more than a hundred locations and a comparable number of transitions. Because of that, both the storage requirements and the running time for the algorithm outlined bellow are negligible.

The weighting procedure for location l is as follows (see below for its rationale):

1. Inactive clocks (in the sense of [DY96], which, roughly speaking, eliminates clocks that are not mentioned in invariants or guards) have 0 weight.

[1] It is not enough to use the location's invariant because the actual range can be restricted by the arriving guards, synchronizations with other components, etc.

2. For the rest of the clocks, an upper and lower bound need to be computed for every location (notation: $lower[\mathsf{l}, x_i]$ and $upper[\mathsf{l}, x_i]$) and every transition (notation: $lower[\mathsf{l} \to \mathsf{l}', x_i]$ and $upper[\mathsf{l} \to \mathsf{l}', x_i]$): initialize those to the values of the respective restrictions. Clocks that are not mentioned have a range of $[0, \infty)$.
3. Then, compute the following fixed point:

   a) For each clock x_i and each location l, let *min* be the minimum of $lower[\mathsf{l}' \to \mathsf{l}, x_i]$, that is, the minimum of the lower bounds of the incoming transitions. Then, $lower[\mathsf{l}, x_i] := \max(lower[\mathsf{l}, x_i], min)$.
   b) For each clock x_i and each transition $\mathsf{l}' \to \mathsf{l}$, let $lower[\mathsf{l}' \to \mathsf{l}, x_i] := \max(lower[\mathsf{l}' \to \mathsf{l}, x_i], lower[\mathsf{l}', x_i])$.

4. For each location and each clock compute its weight as the difference between upper and lower bounds.
5. For each transition $\mathsf{l} \to \mathsf{l}'$ and each clock x_i, if x_i is explicitly mentioned in the transition, use the transition range ($upper[\mathsf{l} \to \mathsf{l}', x_i] - lower[\mathsf{l} \to \mathsf{l}', x_i]$) to compute its weight. Otherwise, use the one from the destination location.

Note that weights in transition include the weight from the destination location, overridden by weight from guards (step 5).

The idea behind steps 2 and 3 is that the constants against which a clock is compared give its spectrum of values. I.e., because of the intersection involved in suc_τ, it will never reach a higher values that its invariant's upper bound. Suppose that only two transitions arrive to l with guards $3 < x_i$ and $5 < x_i$ respectively. If l's invariant states $10 < x_i$, then 10 is the lower bound. But if it were $2 < x_i$, then the lower bound would be 3.

In the algorithm stated above, if a clock is not directly mentioned then its spectrum is given by its values in preceding locations.

During the reachability exploration, composed states are computed on-the-fly. As a by-product, for each discovered state, global transitions T_a are synthesized as follows from transitions[2] $\langle \mathsf{l}_1, a, \psi_1, \alpha_1, \mathsf{l}'_1 \rangle, \ldots, \langle \mathsf{l}_\mathsf{k}, a, \psi_k, \alpha_k, \mathsf{l}'_\mathsf{k} \rangle$ of the k components that took place in the synchronization: $T_a = \langle \langle \mathsf{l}_1 \ldots \mathsf{l}_\mathsf{k} \rangle, a, \psi_1 \wedge \ldots \wedge \psi_k, \alpha_1 \cup \ldots \cup \alpha_k, \langle \mathsf{l}'_1 \ldots \mathsf{l}'_\mathsf{k} \rangle \rangle$.

A *composed clock weight* needs also to be computed as follows:

1. For clocks belonging to automata that took place in the synchronization, take the weights from the transition.
2. For the other clocks, take the weights from the target location.
3. Once each clock has a weight, sort them in descending order.

Each zone that needs to be checked for inclusion comes from the function suc_τ (see line 8 of Algorithm 5.1.1). This function, as its last step, canonizes the DBM to check for emptiness. The canonization process is a variation of Floyd-Warshall minimum path algorithm that takes a DBM in minimal constraint representation form and expands it to its full representation, tightening every bound as much as possible. In order to do so, the linked list is loaded into a square matrix. After the algorithm is finished, the matrix is dumped into a linked list again.

The implementation takes advantage of that: when the square matrix is dumped into a list, bounds are put in the order mandated by the composed weight.

In this way, when IsIncluded?() is called, the bounds have already the proper order.

[2] Assume without loss of generality that only automata 1 to k synchronize with label a.

8.1.4. Threads to Validity

The technique is correct by construction: as clocks are only reordered all of them are still checked, so, in the worst case it can make the checking slower, but not inaccurate.

Regarding the procedure per se, although the rationale for disregarding inactive clocks is quite obvious, the rest of them might need supporting evidence.

To that end, we ran the case studies shown below but only applying the logic that forces inactive clocks to be considered last. The results, omitted to avoid clutter, show that times are worst than the full version of the algorithm, showing that the other considerations indeed speed up the verification. Similar experiments were done, enabling one aspect of the algorithm at the time. The results were still better in the full version.

As was said in Section 8.1.2, there are other threads, which do not seem to occur in the analyzed case studies. Further research is needed to clarify this point.

8.1.5. Case Studies

To validate the technique, we incorporated it into a monoprocessor version of the model checker Zeus and ran a series of experiments. For some of them, a reduced version (created with Obs-Slice [BGO04], a safe model reducer) was also used and is primed in the tables.

The following table summarizes the sizes of the examples used in this article. The experiments were done on a AMD Athlon 64 Processor 3000+ processor with 2 GB of RAM, running Linux 2.6.

Example	#Comps	#Clocks	Reachable states (false)
MinePump	10	10	1 172 911
MinePump'	10	10	131 261
RCS5	8	8	271 771
RCS6	9	9	6 140 255
Conveyor6A	11	12	1 357 200
Pipe6	14	14	80 280
Pipe6'	14	14	24 375
RS-BR	13	14	>1 266 733

Results obtained are summarized in the Table 8.1 (memory usage, being no more than a 2% bigger is not shown in the reordered version).

8.1.6. Conclusions and Open Research Issues

In this section we presented a clock reorder technique that saves time on timed automata model checking, with negligible memory overhead. It should also be mentioned that this optimization is orthogonal to others known, such as zone widening [ZLZZ03], inactive clocks removal [DY96, BBFL03], upper bound abstraction [BBLP04], irrelevant component detection [BGO04], the inclusion check avoidance by hypervolume comparison (introduced below), etc.

Experiments with case studies from the literature show that time saving can go up to a 17%.

Another interesting point to explore is the use of the reorder technique in tandem with CDD-like [BLP+99] data structures, such as the ones presented in Chapter 7. The clock ordering can not only affect the inclusion checks, but might also be used to influence the shape of the diagram, making it less prone to ramifications.

Example	Without Reordering			Savings with Reordering	
	#Checks ($\times 10^6$)	Time (secs)	Mem (MB)	#Checks (%)	Time (secs %)
RCS5 true	14	16	12.52	17.18	4.41
RCS5 false	663	657	58.99	10.30	3.58
RCS6 true	570	476	75.58	10.96	5.46
MinePump' true	28	30	11.90	26.88	8.70
MinePump' false	411	329	32.89	20.72	17.47
MinePump true	2110	1667	93.45	14.01	10.09
MinePump false	14488	11947	254.46	13.51	10.15
Pipe6' true	15	65	26.58	11.06	1.29
Pipe6' false	3	32	20.79	13.11	1.32
Pipe6 false	19	125	54.12	22.97	1.36
Conveyor6A true	268	891	281.04	2.76	2.63
Conveyor6A false	1206	2403	414.69	2.03	4.44
RS-BR true	6984	9385	408.93	13.37	4.95

Table 8.1: Results obtained.

The most important issue however, is to characterize the types of interactions between components that are actually taking place, in order to understand why the threads mentioned in Section 8.1.4 do not materialize and results are still good.

8.2. Hypervolume

8.2.1. Introduction

The basic (conceptual) procedure for forward reachability, shown in Algorithm 5.1.1, is straight forward: insert the initial state in the *Pending* queue and initialize an empty *Visited* set. Then, while *Pending* is not empty, take its next state and check whether the property holds for it. If it does, finish with a "YES" result, otherwise, put it in *Visited*, compute its (timed) successors (one for each outgoing transition). To ensure termination, before putting them in *Pending*, it should be checked that they are not *included* in any other state from *Visited*. In an untimed exploration the check would be simpler: is the same state already *present* in *Visited*? In the timed (symbolic) framework, clock values conform multi-dimensional polyhedra. As such, if a new state is included in an already explored one there is no point in revisiting it, even if it is not exactly equal.

Algorithm 8.2.1 presents a version of Algorithm 5.1.1 where the function IsStateAlreadyVisited?() has been expanded for clarity.

From the outline of the reachability algorithm it should be clear that the inclusion operation between polyhedra is a critical one, being responsible for an important fraction of the total running time. As such, finding ways to speed it up is always a good idea.

Although many optimizations have been developed (see, for example, [BBLP04, BDLY03, Ben01, DY96, Wan03b, BGO04, BOS06a]), the inclusion checking operation, although linear in many cases, has a worst case of $O(n^2)$ where n is the number of clocks in the system. In this section we focus on the *hypervolume* of the above mentioned polyhedra. If it could be easily computed, an $O(1)$ check could be performed prior to the expensive inclusion algorithm: if $hypervolume(A) > hypervolume(B)$ it is impossible for A to be included in B. Of course, if the hypervolume of A is less or equal to B's, then A can be included or not, and the full check needs to be performed.

Computing exact hypervolume for n-dimensional polyhedra is a hard problem, but luckily with

```
1: function ForwardReach(Property φ)
2:     Visited ← ∅
3:     Pending ← {(l_0, z_0)}
4:     while Pending ≠ ∅ do
5:         (l, z) ← next(Pending)
6:         Add((l, z), Visited)
7:         if (l, z) ⊨ φ then return YES
8:         end if
9:         Z_l ← ⋃_{(l,z')∈(Visited∪Pending)} z'
10:        for (l', z') ∈ suc_▷(l, z) do
11:            if ¬ IsIncludedInSet(z', Z_l) then
12:                Add((l', z'), Pending)
13:            end if
14:            if ∃z'' ∈ Pending_l | IsIncluded?(z'', z') then
15:                Delete((l', z''), Pending)
16:            end if
17:        end for
18:    end while
19:    return NO
20: end function
```

Algorithm 8.2.1: Refined forward reachability algorithm.

the traditional data structures used for timed model checking (Difference Bound Matrices) an approximation can be computed very cheaply. This approximation is safe, in the sense that the observations from the previous paragraph still hold (Theorem 8.1 expresses the property formally). A related idea might be comparing the bounding box in each dimension. It has the advantage of being less suceptible to overflow (see Section 8.2.2), but that would require $O(n)$ extra storage and comparisons, instead of $O(1)$.

We take advantage of the hypervolume approximation in two ways: firstly, inclusion checks are avoided in the sense of the preceding paragraphs. Secondly, the *Visited* set is ordered by (approximate) hypervolume. In this way, it is not necessary to check a new state against the complete set, but only against the states that have a bigger (approximate) hypervolume, saving an important number of inclusion checks (see Section 8.2.5).

Neither technique increments the number of visited states. They can be used independently and obtain interesting speedups. Some models benefit more with one of them, and some with the other, with acceleration of up to 19% in our experiments. They can also be combined, although the speedups are not additive, because there is some mutual cancelation. The good news is that there is no need to speculate on which one to use, because using both is generally as good as using the best of them.

Although the idea of approximations is not new (see, for instance the convex-hull abstraction at [Bal96]), they generally lead to approximated answers also (i.e., they might state with certainty that the property is not reached, or that it might –or not– be reached). Ours, however, gives exact answers.

Further, we explain why it does not make sense to order also *Pending* by its hypervolume.

8.2.2. Avoiding Checks by Comparing Hypervolume Approximations

Let's start by defining which approximation to the hypervolume we use. The idea is to compute the hypervolume of the smallest hypercube containing the polyhedron defined by the clocks' values.

Definition 8.1 (Hypervolume Approximation)
Let z be a DBM of a SUA with n clocks. $hvol(z)$ is defined as $\Pi_{1 \leq i \leq n}(const(z[i,0]) - |const(z[0,i])|)$, where $const(x \prec c) = c$.

Note that for the purpose of the hypervolume approximation there is no need to differentiate among $\prec$, and that $const(z[0,i])$, the lower bound, is negative and thus the need for modulus. I.e., $x_1 > 7$ is expressed in DBMs as $z[0,1] = (<, -7)$.

If $z[i,0]$ is $< \infty$, use the maximum constant against which the clock is compared in the SUA (cf. [BBLP04]), plus one.

As can be seen from its definition, the computation of $hvol(z)$ is linear in the number of clocks. However, there is no need to recompute it after every operation that manipulates the zone. It only needs to be calculated as the last step of suc_τ, previous to the inclusion check. The overhead is mild, as the immediate previous operation is usually the transformation of the zone to its canonical form, which is $O(n^3)$.

The approximation is safe, as stated by Theorem 8.1.

Theorem 8.1 (*hvol* is a Safe Approximation)
If $hvol(z_1) > hvol(z_2)$ then $z_1 \not\subseteq z_2$.

Proof

Let's assume $hvol(z_1) > hvol(z_2) \wedge z_1 \subseteq z_2$. As the operation IsIncluded?() is sound and complete w.r.t. $\subseteq$, it means that $(\forall i,j)\ 0 \leq i,j \leq n, i \neq j \implies const(z_1[i,j]) \leq const(z_2[i,j])$. Then, as $hvol(z_1)$ and $hvol(z_2)$ are both products of the same quantity of positive terms, $hvol(z_1)$ is the product of positive numbers which are all less or equal to the corresponding ones in $hvol(z_2)$, contradicting the possibility of $hvol(z_1)$ being greater than $hvol(z_2)$.

□

Care must be taken when computing $hvol(z)$ as to not overflow the capacity of the container integer variable. Which type of integer variable to use for storing the *hvol* of a zone has consequences in both memory overhead and precision. The lower the number of octets reserved for the *hvol*, the sooner it will saturate, not allowing to avoid some inclusion checks. On the other hand, if too many octets are used, the memory overhead can be considerable. The exact hypervolume would require $O(\log \Pi_{1 \leq i \leq n}|C_i|)$, where C_i is the biggest constant compared against the x_i clock in the system. As with many others time-vs-memory trade offs, experimentation should be used to find a convenient balance.

Note that the overhead depends on the implementation of DBMs. If a proper matrix is used, a `long int`, which is 8 octets on 32 bits machines, usually provides a good amount of check saving and requires little space compared to the DBM itself. On more sophisticated representations which leverage on Minimal Constraint Representation [LLPY03] to use a variant of linked lists of bounds –like our packed RDBMS (see Section 3.3)– the overhead can be variable. Our experimentation shows an average of 2% extra memory. Although Section 8.2.5 shows the experimental results, let's suppose we are dealing with a system with ten clocks (a conservative assumption). A proper DBM will have a hundred bounds. Being conservative, assume that the minimal representation has approx. 30% of the bounds. For 30 bounds and 12 bits per bound (usually enough to represent constants on the hundreds), the 360 bits can be packed in 12 `long int`. For these figures, an extra `long int` represents less than 9% of extra memory.

Although hypervolume approximation resembles the convex-hull abstraction [Bal96], there is a very important difference. Ours is an exact technique, meaning that the answer to the reachability question is responded *yes* or *no* with certainty. In convex-hull, on the other hand, zones corresponding

to the same location are joined to their convex-hull over approximation. If the property is not reached, then the answer is precise, but if it is, then the answer is *maybe*.

8.2.3. Sorting *Visited*

As can be seen in Algorithm 8.2.1, when a new state (l, z) is found, the $Visited_{\mathsf{l}}$ set (i.e., the restriction of *Visited*[3] to states that have l as location) has to be fully explored, comparing the new zone against all the ones contained in that set.

If each zone in *Visited* has an *hvol*, then it can be turned into a priority queue, where the zones with greater *hvol* are checked first. Let z_n be the zone of the new state and z_q be the zone of $next(Visited_{\mathsf{l}})$. $hvol(z_q)$ is bigger or equal than $hvol(z)$ for every $z \in Visited_{\mathsf{l}}$. So, if $hvol(z_n)$ is greater than $hvol(z_q)$, then, because of Theorem 8.1, z_n is not included neither in z_q, nor in the rest of $Visited_{\mathsf{l}}$. In consequence, there is no need to continue checking, thus reducing the number of inclusion checks performed, as can be seen in the experimentation.

A problem of implementing the above mentioned technique with a traditional priority queue is that iteration is done by the successive elimination of *next* elements, thus requiring to re-insert them afterwards. For a queue with k elements, the total cost with, e.g., a heap, is $O(k \log k)$. The memory management involved in removing and adding elements can make the constants considerable, importing a noticeable overhead that can easily counter the gain from the inclusion checks avoided.

To overcome this problem, we chose a van Emde Boas tree [vEBKZ77], which permits nondestructive iteration. It is also convenient from a theoretical point of view: visiting the first k' elements costs $O(k' \log \log k)$. Actually van Emde Boas trees provide all of their operations in $O(\log \log k)$, at the penalty of only supporting integer numbers from a fixed interval as keys. Our experience with it was that although it can be quite difficult to implement, it provides very good performance.

The resulting procedure is shown in Algorithm 8.2.2.

```
function IsIncludedInOrdederSet?(DBM z, DBM ordered set Z)
    for all z' ∈ Z do
        if hvol(z) >hvol(z') then
            return NO
        else if IsIncluded?(z, z') then
            return YES
        end if
    end for
    return NO
end function
```

Algorithm 8.2.2: Inclusion checking algorithm (zone in ordered set of zones).

Having $Visited_{\mathsf{l}}$ sorted by *hvol* is not only useful when the new zone is not included. If it is, as the bigger zones are checked first, there is a good chance that the detection occurs earlier (for instance, universal zones are always the first to be checked, whereas in a traditional implementation of $Visited_{\mathsf{l}}$ they could be "buried" deep into the set).

It should be noted that it makes no sense to check if a new state (l', z') includes one (l', z'') in $Visited_{\mathsf{l}}$ (contrary to $Pending_{\mathsf{l}}$ which is checked in line 14 of Algorithm 8.2.1), because in case it is, (l', z'') still cannot be removed, as it might be part of a trace to the target state.

Section 8.2.5 shows the time and memory results for the implementation of the above mentioned techniques in the model checker Zeus. Before that, in Section 8.2.4, we explore the question of whether

[3] Although we use a unified storage of $Visited \cup Pending$ as proposed in [BDLY03], separate indexes allows us to traverse them independently.

Example	Components	Clocks	Reachable locations
MinePump	10	10	1428
RCS5	8	8	1617
Conveyor6A	11	12	31443
FDDI8	15	23	4608

Table 8.2: Examples sizes.

it is also worth sorting *Pending*.

8.2.4. Sorting *Pending*. Worth it?

At first sight the idea of sorting *Pending* by decreasing *hvol* sounds appealing: suppose that both z and z' are in *Pending*, and that $hypervolume(z) > hypervolume(z')$. As suc_τ is monotonic, it makes sense to compute $\hat{z} = suc_\tau(z)$ before $\hat{z'} = suc_\tau(z')$ because $\hat{z}$ is bigger that $\hat{z'}$. Chances are that $\hat{z'}$ might be included in $\hat{z}$, avoiding the exploration of a new zone.

An interesting aspect of sorting *Pending* that way is that it *seems* conservative. It *seems* that in case it didn't improve things, they will not get worse, i.e., no more zones will be generated.

To test these ideas, we changed the *Pending* FIFO queue into a priority queue sorted by *hvol*, and implemented it also as a van Emde Boas tree. Unfortunately, results were not positive, leading to more zones found (and thus more total time and memory) for many case studies, even ones that generated the complete state space.

The problem is that when locations are considered into the equation, things get more complicated than the intuition presented in previous paragraphs. Suppose there are (l_1, z_1) and (l_2, z_2) in *Pending* (in that order), with z_1 being $x < 10$ and z_2 being $x < 12$. Also, assume that both l_1 and l_2 have a transition to l_3 (which has a *true* invariant), but the second with an $x > 5$ guard while the first imposes no restriction.

In a FIFO exploration (l_1, z_1) will be explored first, leading to the discovery of (l_3, z_3), with z_3 being $x < \infty$. When (l_2, z_2) is expanded, the new state, with zone $z'_3 = 5 < x < \infty$, will already be included. On the other hand, if *Pending* was ordered by hypervolume, (l_2, z_2) would be explored first, leading to the discovery of (l_3, z'_3), which in turn will be put into *Pending* and explored before (l_1, z_1). When this state gets its turn, (l_3, z_3) will be generated, but the inclusion check will fail, thus creating a new zone to be explored.

Next section shows the experimental evidence that backs the claims made in Sections 8.2.2 and 8.2.3.

8.2.5. Case Studies

To validate the proposed techniques, we incorporated them into a monoprocessor version of our Zeus model checker and ran a series of experiments against well known case studies from the literature (detailed in Appendix A). For some of them, a reduced version (created with ObsSlice [BGO04], a safe model reducer) was also used and is primed in the tables. Each of them comprises two versions: the one where the property is reached (true) and the unreachable one (false).

Table 8.2 summarizes the sizes of the examples used in this section. The experiments were run on a Intel Pentium IV 3.0 GHz processor with 2 GB of RAM, running the Linux 2.6 operating system.

Table 8.3 shows the results obtained with each method independently and Table 8.4 the combination of both. The first columns report the time, memory and number of inclusion checks for the

Example	Standard			With *hvol*		*Visited* priority queue		
	Time (secs)	Mem (MB)	#checks ($\times 10^6$)	Time (secs)	#checks ($\times 10^6$)	Time (secs)	Mem (MB)	#checks ($\times 10^6$)
MinePump' true	46	7	10.5	43 (-7%)	8.9 (-15%)	43 (-7%)	7.8 (+11%)	9.2 (-12%)
MinePump' false	527	19	206.0	475 (-10%)	173.8 (-16%)	465 (-12%)	21.2 (+12%)	180.6 (-12%)
MinePump true	2807	56	1124.3	2542 (-9%)	969.7 (-14%)	2444 (-13%)	62.1 (+11%)	959.3 (-15%)
MinePump false	20580	152	4348.9	18353 (-11%)	3051.1 (-30%)	17425 (-15%)	163.2 (+7%)	3030.2 (-30%)
RCS5 true	31	7	10.2	23 (-26%)	5.5 (-46%)	28 (-10%)	7.7 (+10%)	9.1 (-11%)
RCS5 false	1039	34	341.3	952 (-8%)	286.4 (-16%)	955 (-8%)	34.6 (+2%)	311.7 (-9%)
Conveyor6A true	1385	183	163.6	1280 (-8%)	104.8 (-36%)	1338 (-4%)	198.0 (+8%)	138.5 (-15%)
Conveyor6A false	4142	280	998.8	3368 (-19%)	567.1 (-43%)	3600 (-13%)	306.4 (+9%)	762.7 (-23%)
FDDI8 true	488	20	0.1	488 (0%)	0.1 (0%)	486 (-<1%)[4]	21.0 (+5%)	0.1 (-<1%)
FDDI8 false	7642	120	18.2	7645 (+<1%)	18.2 (0%)	7629 (-<1%)	122.0 (+2%)	18.2 (-<1%)

Table 8.3: Results obtained for each method.

standard version, and then the time and number of inclusion checks for each optimization, with the percentage in parenthesis (negative values for saving, positive for increase). Memory is not reported for the first optimization because the overhead was always less than 2% (consistently with the reasoning already mentioned in Section 8.2.2). The overhead of the second comes from the van Emde Boas tree, which requires many pointers.

It should be noted that in the combined method, although not direct check is saved by *hvol*, many inverse ones (see line 14 of Algorithm 8.2.1) can still be avoided.

As can be seen in Table 8.3, *hvol* features time savings of up to 19% (not counting *RCS5* true, because it already takes a very short time and is only presented for completeness), and sorting *Visited* of up to 15%. The first one has negligible memory overhead, while the second uses an extra of around 10-12%. There is no direct relationship between the number of checks saved and the speedup. This is because each case studies has different sized matrices and for each check saved, the number of cell matrices to be compared differs.

FDDI8 is an interesting case study because, as no inclusion check could be avoided with *hvol*, the difference in times measures pure overhead, showing that the method is very light.

Table 8.3 also shows that different case studies benefit the most from different techniques. If memory is a premium, clearly *hvol* is convenient, but if some memory can be spent in order to obtain earlier results, then a decision should be made among the two. Also, they can be combined. As shown in Table 8.4, with the exception of *FDDI8*, the combined method is never worse than the worse of the optimizations (see for instance *Conveyor6A*). Sometimes it is as good as the best of them (*MinePump'*, *MinePump* true, *RCS5* true) and sometimes better (*MinePump* false, *RCS5* false).

Both the second and the combined method have indeed a memory overhead that seems, in terms of percentages, on the same order of magnitude as the time saved. However, in cases like *MinePump* false in Table 8.4, it can be seen that the 16% saving of time translates to almost one hour of almost

[4]Marginal improvements, not seen in figures because of rounding.

Example	Both methods		
	Time (secs)	Mem (MB)	#checks ($\times 10^6$)
MinePump' true	43 (-7%)	7.8 (+11%)	8.3 (-21%)
MinePump' false	465 (-12%)	21.2 (+12%)	181.3 (-12%)
MinePump true	2436 (-13%)	62.1 (+11%)	955.7 (-15%)
MinePump false	17388 (-16%)	163.2 (+7%)	2348.4 (-46%)
RCS5 true	23 (-26%)	7.5 (+10%)	8.0 (-22%)
RCS5 false	926 (-11%)	34.6 (+2%)	286.7 (-16%)
Conveyor6A true	1295 (-6%)	198.0 (+8%)	115.9 (-29%)
Conveyor6A false	3470 (-16%)	306.4 (+9%)	569.3 (-43%)
FDDI8 true	487 (-<1%)	21.0 (+5%)	0.1 (-<1%)
FDDI8 false	7652 (+<1%)	122.0 (+2%)	18.2 (-<1%)

Table 8.4: Results obtained with both methods combined.

six, at the cost of 7% more memory, which only amounts to less than 12 extra MB of RAM.

8.2.6. Conclusions and Future Work

In this section we presented two techniques based on approximating the hypervolume of the polyhedra that represents the valuations of clocks in timed automata model checking. Although the hypervolume is approximated, both techniques give exact answers.

The first is based on avoiding inclusion checks when the approximate hypervolumes makes the inclusion impossible. The other, in sorting the *Visited* set according to the approximate hypervolumes, avoiding to traverse some parts of it while checking for included zones. The second is only possible thanks to a very optimized implementation of a van Emde Boas tree [vEBKZ77], but the first is quite simple.

These techniques can be used independently and obtain interesting speedups. According to our experiments, some models benefit more with one of them, and some with the others, with acceleration of up to 19 and 15% respectively. The first one has negligible memory overhead, the second, a moderate one (10-12%).

They can also be combined, although the speedups are not additive, because there is some mutual cancellation. The good news is that there is no need to speculate on which one to use, because using both is generally as good as using the best of them.

It should be noted that the techniques are very unobtrusive, in the sense that they are orthogonal to many other optimizations such as [BBLP04, BDLY03, Ben01, DY96, BGO04, BOS06c], allowing to use all of them together.

Also, we showed that applying the same ideas to the *Pending* queue can have negative impact. In order to reverse that, *Pending* should be separated by location, and then sorted, but that will increase the cost of adding newly discovered zones to it. Experimentation with different data structures towards that end is a yet-to-be-explored area.

Although we chose a van Emde Boas tree to sort the *Visited* queue, some less modular yet simpler implementations –based on linked lists of states, with pointers marking insertion places– are possible. This trade-off should be revisited to see if some of the overhead can be avoided.

To pursue further in this line of research, it would be interesting to analyze which topological characteristics of the model influence in each method's performance. A consequence of this could be an on-the-fly detection method, that switches between them.

Chapter 9

VInTiMe

This chapter is based on [CdCF+06], where the VInTiMe tool is presented in more detail (while [AGB+04] gives the theoretical background). VInTiMe can be download from http://lafhis.dc.uba.ar/vintime as an Eclipse[1] plugin.

The tool represents the hard work of the following bright individuals:

- Lucía Cavatorta
- Guido de Caso
- Andrés Ferrari
- Víctor Braberman
- Diego Garbervetsky
- Nicolás Kicillof
- Alfredo Olivero
- Alejandra Alfonso

We would like to thank the IBM Eclipse Innovation Grants Program, who were kind enough to fund the plugin development.

9.1. Introduction

In order to verify real world applications, usually the following steps should be followed:

- The SUA should be precisely described.
- System requirements must be specified.
- A (usually time consuming) verification should be executed.
- If requirements do not hold, appropriate counterexamples must be obtained in order to fix the system.

VInTiMe (Verifier of INtegrated TImed ModEls) [AGB+04, CdCF+06] is a suite of tools that combine high-level expressive power, unassisted property-preserving model-reduction and low-level distributed model checking power to describe and verify complex real-time system designs and their properties. VInTiMe comprises the integration of the tools Lapsus, VTS, ObsSlice and Zeus. As a toolset, it covers the full specification and verification cycle. This is actually the result of a decoupled approach in which autonomous tools with clear interfaces were developed independently. They were

[1] According to http://www.eclipse.org, Eclipse is an "open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the life cycle". When the term is used in this chapter, we are actually referring to the IDE.

unified under a single framework once they became mature enough, using timed automata as a *lingua franca* between them.

Lapsus [BF99, Bra00] translates real-time system designs based on fixed-priority scheduling into timed automata. Succinctly, it uses Worst-Case Completion Times (WCCT) and Best-Case Completion times (BCCT) –which can be calculated using analytical techniques provided by these theories– to build an abstract and analyzable model of the system as a collection of timed automata. We use those timed automata to analyze complex properties involving coordination of a set of tasks.

Once Lapsus' job is done, VTS [ABKO04] comes into play. Using simple graphical patterns, the designer can define interesting properties over the SUA. More precisely, VTS is a notation for expressing real-time requirements in a visual and friendly –yet powerful– language, by means of negative scenarios. A VTS scenario is basically an annotated partial order of relevant events, denoting a (possibly infinite) set of matching time-stamped executions.

VTS is meant to existentially predicate on system executions. That is, it is used to state a simple though relevant family of questions of the form "Is there a potential run that matches this generic scenario?". When interpreted as negative scenarios, these questions can express infringements of safety or progress requirements, which turn out to be decidable. The tool translates a scenario into a timed automaton (observer) that recognizes matching runs. This automaton is composed with the SUA in order to check whether a violating execution is reachable in that behavioral model.

It should be noted that the tools presented so far tackle the usability issue by simplifying the construction of formal models of both timed systems and properties over them. There is still a scalability problem to deal with, and that is the goal of the two remaining components of VInTiMe: they deal with the state explosion problem in orthogonal ways.

ObsSlice [BGO04] is an optimization tool for the verification of networks of timed automata using virtual observers (see Section 2.7), the same kind of observers that VTS produces. It automatically discovers the set of modeling elements that can be safely ignored at each location of the observer by synthesizing behavioral dependence information among components. ObsSlice is fed with a network of timed automata and generates a transformed network which is equivalent to the one provided, as far as the properties are concerned.

ObsSlice automatically derives a subset of components which is enough to perform the verification process. That is, one of the most appealing aspect of this approach is that generally only a small subset of a system's components is really needed to perform model checking, dramatically reducing the cost of this step. Hence, the complexity of our approach mainly depends on the number of components involved in the requirement instead of the size of the complete model, which is the case in previous works on automatic verification of real time applications.

Contrary to other approaches that use schedulers, Lapsus modeling of real-time tasks helps ObsSlice reduce the amount of components that are relevant to verify a given property. If schedulers were used, the resulting system would be highly coupled, making optimization harder [Bra00].

Finally, Zeus is the model checker dealt with in this very thesis.

9.2. Features

The development of real-time systems requires specification and implementation frameworks, usually realized as separate software tools, which forces designers to constantly switch between them. The integration of VInTiMe as an Eclipse plug-in greatly simplifies this process. Designers can have their systems both specified and implemented entirely within the Eclipse platform.

Eclipse also provides VInTiMe users with additional features such as the ability to have their verification projects saved in CVS repositories or exported to ZIP files.

In order to support the integration of the tools described in the previous section, the VInTiMe Eclipse plug-in offers a set of editors for each of the formalisms involved.

The verification process begins with the creation of a new VInTiMe project to contain the different elements necessary to model the SUA and to guide its verification.

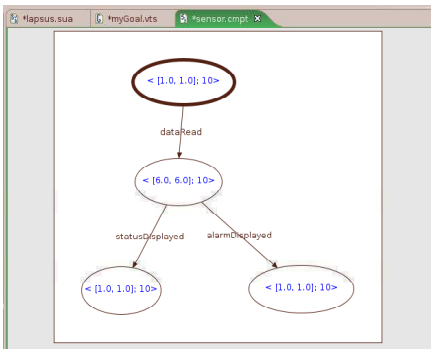


Figure 9.1: A Lapsus component

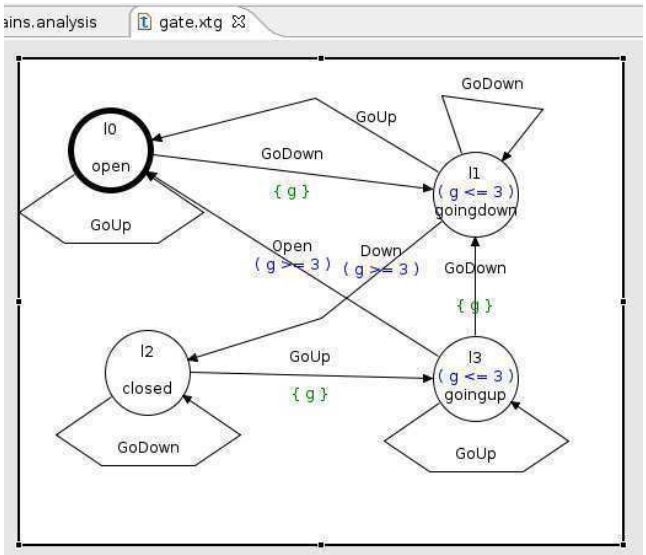


Figure 9.2: A TA component

Since real-time systems are usually made up of several interacting components, the designer starts by constructing each of them independently. Main system components are modeled using Lapsus tasks (Fig. 9.1), while environment and user defined connectors can be directly modeled with TA (Fig. 9.2).

Once the components are defined, the user can create a SUA by dragging together instances of

them. Making component construction and composition separate steps of the process maximizes reusability of system parts.

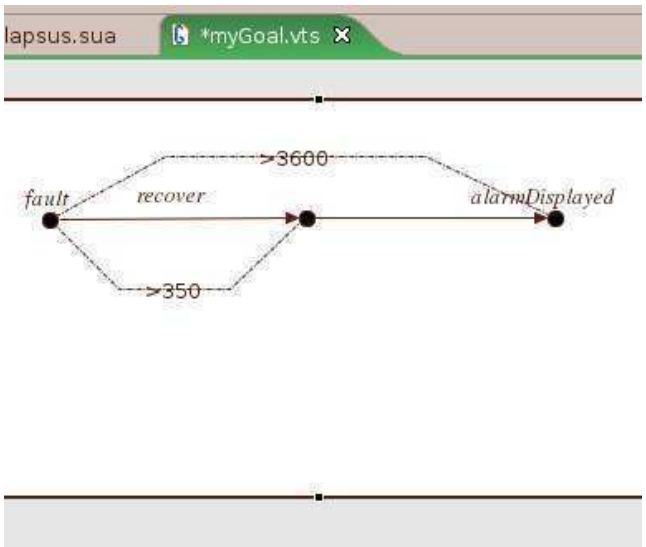


Figure 9.3: A VTS scenario

After the SUA has been defined, users can express requirements over it in the form of VTS patterns, as shown in Fig. 9.3. Since these scenarios can contain variables, an instantiation phase is required: users drag goals and SUAs together while a wizard helps them bind variables.

This binding of goals to SUAs is called an *Analysis*. Fig. 9.4 shows the Analysis Editor.

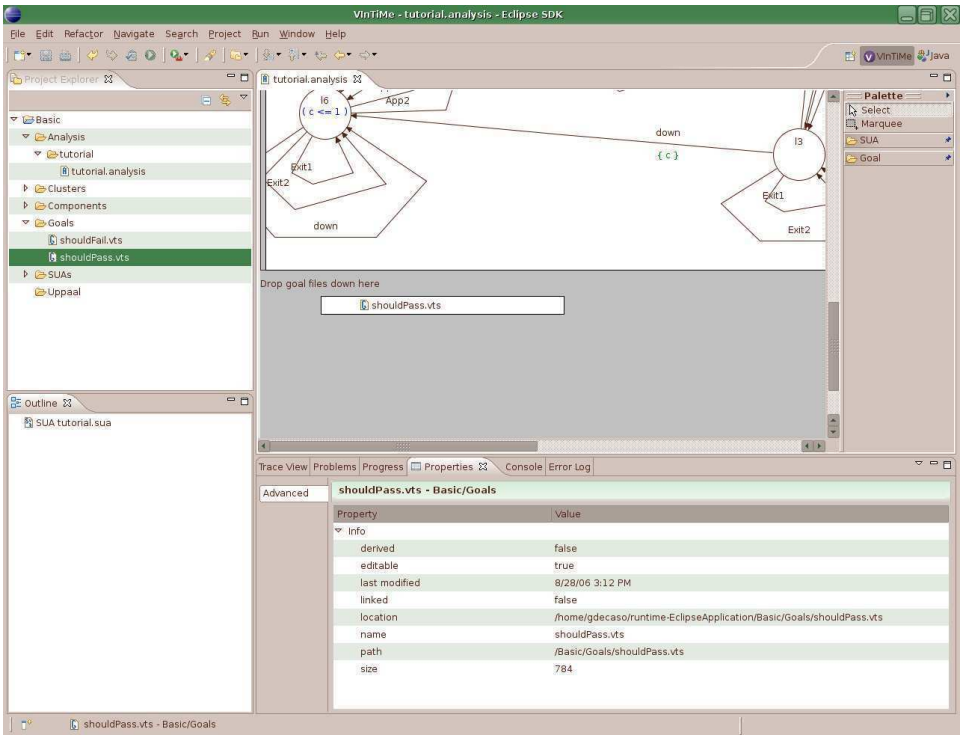


Figure 9.4: Analysis Editor

This is the final step in the design stage, now the verification phase can begin. From the Analysis Editor users can call ZEUS to perform a distributed model check of a goal on a cluster. A cluster is a network of workstations that can be graphically defined in the bundled Cluster Editor (see Fig. 9.5)

and selected later when the verification is launched (see Fig. 9.6).

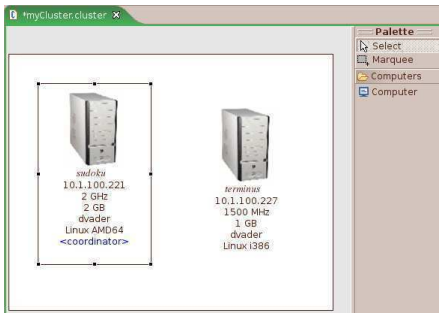


Figure 9.5: Cluster Editor

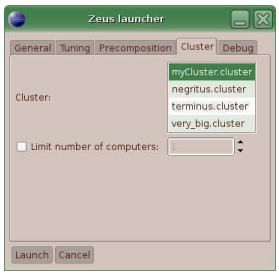


Figure 9.6: Launching Zeus

If the SUA can produce a behavior expressed by the (negative) goal, a trace showing how it can be reproduced is presented in a Trace View (Fig. 9.7). Otherwise, the goal is valid and no trace is found. As a visual aid, a color cue is used to mark goal states.

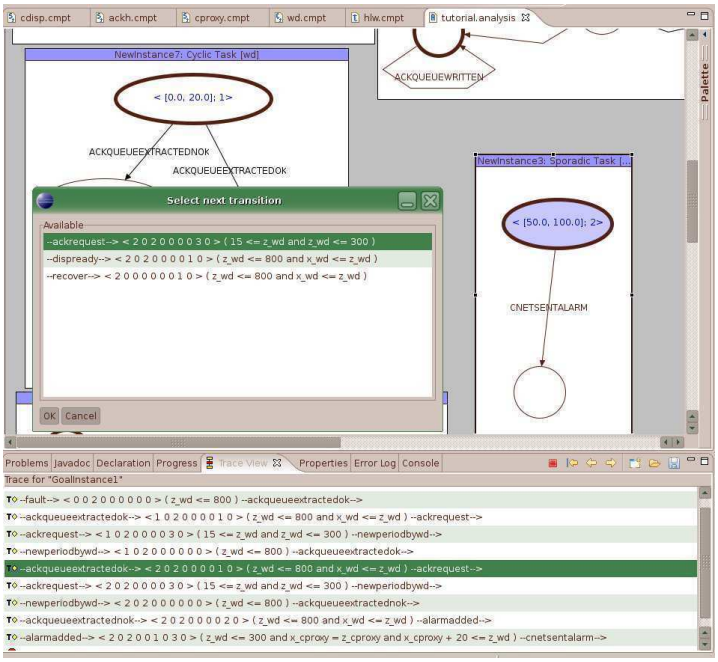


Figure 9.7: Trace View

The SUA can be traversed by following a trace. While doing so, current states and transitions are highlighted appropriately. As a result of applying ObsSlice during verification, SUA components are dynamically enabled or disabled in a trace. Our tool automatically minimizes disabled SUA

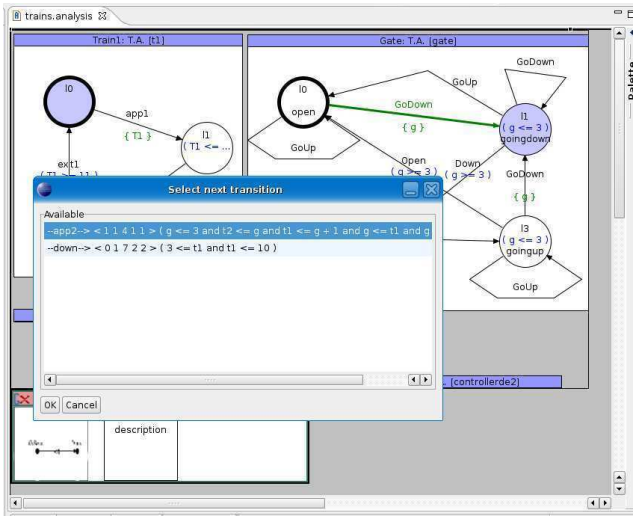


Figure 9.8: Users select how they want the trace to continue

components during trace traversal and restored when they are enabled again. This greatly simplifies the task of understanding error traces, as only relevant components need to be analyzed. Traces can also be manually modified, or generated from scratch in order to allow the user to simulate additional interesting behaviors, as shown in Fig. 9.8.

Chapter 10

Conclusions and Perspective

We finish this thesis by browsing through each chapter to find the most relevant conclusions and open research questions.

Chapter 4 analyzes the distribution of the backwards algorithm. It can be summarized as a succession of obstacles, workarounds and modest practical achievements. However, it presents hard efforts based on sound ideas, leaving testimony of how hard the problem at hand is. Although far from the expected linear speedups, the results allow to conclude that, in order to achieve something close to it, a unit of distribution finer that the location is needed. Although still preliminar, some work we are currently undertaking in that area seems promising: parallelize expensive matrix operations on multicore processors, instead of using each core to run an independent *capsule*.

Chapter 5 opposes Chapter 4 not only in that it is based on a different algorithm (forward, in this case), but also in that it opens rather than closes, an area of research. Although the machinery put in place to migrate workload (cf. Section 5.3) is highly sophisticated and its results are promising, there is still a lot of work to do, and ideas yet to be explored. We surveyed those on Section 5.5: hybrid approaches, partitioning based on clock ranges and auto-adjusting of tunables.

In Section 5.5 we pointed out that a number of variables could be tuned to affect the speed of the verification, and that there does not seem to be a clear winning set of options. Even without leaving the boundaries of this same thesis, we found again examples of that in the monoprocessor settings of Chapters 7 and 8.

This landscape triggers a fundamental question that arises repeatedly when working on timed automata model checking: is there any *distinguishable* characteristic of the model that correlates directly with some techniques performing better than other? Even more, if such a characteristic exists, is it distinguishable *by humans*?

If that were the case, and we could indeed assert which are the specific properties of the models that make them react better to some optimizations, then, maybe the model checker could be programmed to determine that automatically. We don't seem to be close to that point. Yet, we might be able to make progress, by changing the question. Why don't we ask if there is a specific characteristic that can be spotted *by computers*?

Perhaps it's time to use machine learning to try to correlate speedups with topology. Is it too early to give up on humans trying to find out a correlation? Maybe yes, but it is for sure a very difficult task: all known optimizations are based on complex ideas. While some of them are oblivious (e.g., [YPD94, DY96, BGO04]), others are useful in only particular scenarios ([Bal96, BOS06b]). Some others (e.g., [BBFL03, ZLZZ03]) are so complex that the reason they work cannot be explained intuitively. And there is still a class of optimizations that make sense only if some supporting data structure is used (e.g., [Ben01, BDLY03]).

Further, partial order reduction techniques (e.g. [LNZ05]) can be hindered by a distributed algo-

rithm. So orthogonality is not that easy to achieve.

But not all of the complexity comes from optimizations. Models bring their share also. On one hand, they shape can no longer be easily seen. They are built from the synchronization of many components, quickly reaching an important number of states and transitions. On the other, many of those models are not built by hand anymore. While in some cases that imposes regularities to the automata, in others it just means huge components, with difficult semantics.

In the same pool can be considered the different tunables that Zeus uses. For instance, if the load pattern changes very frequently, a low Δt_m and $\#s$ are in order (recall Section 5.4). But overhead might be increased if that is not the case.

How to know all that, *a priori*? Even more, how to expect for an average verification engineer to take all of those decisions correctly before-hand?

Given all the above mentioned difficulties is that it becomes appealing to automate the optimization decision-making process. A naïf approach might try to cluster control graphs using known techniques (e.g., [vD98]) and try to pair classes of graph with verification settings, based on a knowledge-base of experience. It is to be expected that good results might only appear once the "graph decorations" (i.e., time restrictions) start to be considered. If the approach is clustering, maybe the work in timed regular expressions [ACM02] comes out handy, as a way to succinctly describe particular timing behaviours.

While comparing our work with others, in Section 6.1, we discovered that interesting speedups might be obtained –in certain cases–, not because of distribution, but actually as a byproduct of it. As most inclusion checks seem statistically destined to fail, skipping some of them might save time with little probabilities of repeated work. We believe this phenomenon explains part of the good results obtained by some other researchers, and poses an interesting question: could similar results be replicated on a monoprocessor setting?

In Chapter 7 we researched alternative data structures. In particular, rCDDs, a BDD-like representation. We managed to reduce time at the expense of some memory, and found that detecting repeated subtrees was a key issue: if memory consumption could be reduced, regions could start to be used, thus leading to less explored states. Much work is pending in that area.

Chapter 8 proposes two techniques that can be used in the monoprocessor version. One is based on predicting clock ranges, the other on using single integer comparisons (based on zone hypervolume) to avoid $O(n^2)$ inclusion checks. The last one, given its simplicity, has all the characteristics needed to become a *de facto* standard mechanism in the TA model checking tools, like active clock reduction [DY96].

Clock ranges, however, present a lot of open ends. Not only improving the prediction has to be addressed, but also understanding the details of why such an incomplete prediction works so well.

We always aimed at usable techniques. So we tried not only to implement ideas into tools, but also to integrate tools into frameworks. This is what VInTiMe is about. In Chapter 9 we described a verification toolset that "combines high-level expressive power, unassisted property-preserving model-reduction and low-level distributed model checking power to describe and verify complex real-time system designs and their properties."

To conclude, a note on set inclusion. Not that many people have built a model checker. Fewer have done it twice. This is the subset of humans we are included in, having built both the backwards (surveyed in Chapter 4) and forward (Chapter 5) versions of Zeus.

Bibliography

[ABKO04] Alejandra Alfonso, Víctor Braberman, Nicolas Kicillof, and Alfredo Olivero. Visual timed event scenarios. In *Proc. of the 26th ACM/IEEE International Conference on Software Engineering*. ACM Press, 2004.

[ABL98] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. Model checking via reachability testing for timed automata. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*, pages 263–280, 1998.

[ACD+92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 157–166, Phoenix, Arizona, 1992.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed Regular Expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFH91] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.

[AGB+04] Alejandra Alfonso, Diego Garbervetsky, Víctor Braberman, Alfredo Olivero, Nicolás Kicillof, and Fernando Schapachnik. Vintime: Combining high-level finesse with low-level muscle to verify real-time systems. In *First International Conference on Principles of Software Engineering, PRISE 2004*, Buenos Aires, Argentina, November 2004.

[AHU89] Aho, Hopcroft, and Ullman. *Data Structures and Algorithms*. Addison Wesley, 1989.

[Amd67] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference*, volume 30, pages 483–485. Spring Joint Computing Conference, 1967.

[AT02] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In *RT-TOOLs*, 2002.

[AVL62] Georgii M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, (146):263–266, 1962.

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 52, Washington, DC, USA, 1996. IEEE Computer Society.

[BBFL03] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 254–277, Warsaw, Poland, apr 2003. Springer.

[BBLP04] G. Behrmann, P. Bouyer, K. Larsen, and R. Pelnek. Lower and upper bounds in zone based abstractions of timed automata. In *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 312–326. Springer Verlag, 2004.

[BBS01] Jiri Barnat, Lubos Brim, and Jitka Stríbřná. Distributed LTL model-checking in SPIN. In Matthew B. Dwyer, editor, *Proc. of the 8th International SPIN Workshop*, pages 200–216, Toronto, Canada, 2001.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDHGS00] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer-Aided Design*, pages 390–404, 2000.

[BDLY03] Gerd Behrmann, Alexandre David, Kim G. Larsen, and Wang Yi. Unification & sharing in timed automata verification. In *SPIN Workshop 03*, volume 2648 of *LNCS*, pages 225–229, 2003.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. of the 10th Intl. Conf. CAV '98*, volume 1427 of *LNCS*, pages 546–550. Springer-Verlag, 1998.

[Beh02] Gerd Behrmann. A Performance Study of Distributed Timed Automata Reachability Analysis. In *Workshop on Parallel and Distributed Model Checking, afiliated to CONCUR 2002 (13th International Conference on Concurrency Theory)*, volume 68 of *ENTCS*, Brno, Czech Republic, August 2002. Elsevier.

[Beh05] Gerd Behrmann. Distributed reachability analysis in timed automata. *International Journal of Software Tools for Technology Transfer*, 7(1):19–30, February 2005.

[Ben01] Johan Bengtsson. Reducing memory usage in symbolic state-space exploration for timed systems. Technical Report 2001-009, Department of Information Technology, Uppsala University, 2001.

[BF99] Víctor Braberman and Miguel Felder. Verification of real-time designs: Combining scheduling theory with automatic formal verification. In *Software Engineering - ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 521–525, Touluse, France, September 1999. Springer-Verlag.

[BGO02] Víctor Braberman, Diego Garbervetsky, and Alfredo Olivero. Improving the verification of timed systems using influence information. In *TACAS 2002, held as part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002*, volume 2280 of *LNCS*, pages 21–36, Grenoble, France, April 2002. Springer-Verlag.

[BGO04] Víctor Braberman, Diego Garbervetsky, and Alfredo Olivero. ObsSlice: A timed automata slicer based on observers. In *Proc. of the 16th Intl. Conf. CAV '04*, LNCS. Springer Verlag, 2004.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *Computer Aided Verification*, volume 1855 of *LNCS*, pages 216–231. Springer-Verlag, 2000.

[BKO05] Víctor Braberman, Nicolas Kicillof, and Alfredo Olivero. A scenario-matching approach to the description and model checking of real-time properties. *IEEE Transactions on Software Engineering*, 31(12):1028–1041, 2005.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243. Springer-Verlag, 1995.

[BLO02] Víctor Braberman, Carlos López Pombo, and Alfredo Olivero. On improving backwards verification for timed automata. In *TPTS 2002, satellite event for the Joint European Conference on Theory and Practice of Software, ETAPS 2002*, volume 65 of *ENTCS*, Grenoble, France, April 2002. Elsevier.

[BLOS07] Víctor Braberman, Jorge Lucángeli, Alfredo Olivero, and Fernando Schapachnik. Hypervolume approximation in timed automata model checking. In Jean-François Raskin and P.S. Thiagarajan, editors, *International Conference on Formal Modelling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer, oct 2007.

[BLP+99] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using Clock Difference Diagrams. In *Computer Aided Verification*, pages 341–353, 1999.

[BLW01] Benedikt Bollig, Martin Leucker, and Michael Weber. Parallel model checking for the alternation free μ-calculus. In 7th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *LNCS*, pages 543–558, 2001.

[BOS02] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Zeus: A distributed timed model checker based on Kronos. In 1st *Workshop on Parallel and Distributed Model Checking, affiliated to CONCUR 2002 (*13th *International Conference on Concurrency Theory)*, volume 68 of *ENTCS*, Brno, Czech Republic, August 2002. Elsevier.

[BOS04a] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. On-the-fly workload prediction and redistribution in the distributed timed model checker ZEUS. In 3rd *International Workshop on Parallel and Distributed Methods in verifiCation, affiliated to CONCUR 2004 (*15th *International Conference on Concurrency Theory)*, ENTCS, London, UK, September 2004. Elsevier.

[BOS04b] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. On-the-fly Workload Prediction and Redistribution in the Distributed Timed Model Checker Zeus. In Lubos Brim and Martin Leucker, editors, *Workshop on Parallel and Distributed Methods in verifiCation, affiliated to CONCUR 2004 (*15th *International Conference on Concurrency Theory)*, ENTCS, London, UK, September 2004. Elsevier.

[BOS05] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Issues in Distributed Model-Checking of Timed Automata: building ZEUS. *International Journal of Software Tools for Technology Transfer*, 7:4–18, feb 2005.

[BOS06a] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Avoiding inclusion checks via clock reordering in timed automata model checking. Technical report, Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina, September 2006.

[BOS06b] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Dealing with practical limitations of distributed timed model checking for timed automata. *Formal Methods in System Design*, 29:197–214, Sep 2006.

[BOS06c] Víctor Braberman, Alfredo Olivero, and Fernando Schapachnik. Optimizing timed automata model checking via clock reordering. In *27th IEEE International Real-Time Systems Symposium, Work in Progress Session*. IEEE, nov 2006.

[Bra00] Víctor Braberman. *Modeling and Checking Real-Time Systems Designs*. Ph d. thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2000.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[CdCF+06] Lucía Cavatorta, Guido de Caso, Andrés Ferrari, Víctor Braberman, Diego Garbervetsky, Nicolas Kicillof, Alfredo Olivero, and Fernando Schapachnik. A toolsuite for the verification of realtime systems in eclipse. In *ETX 2006: OOPSLA workshop on eclipse Technology eXchange*, page (in press). ACM, ACM, 2006.

[Cou78] Patrick Cousot. *Methodes Iteratives de Construction et D'Aproximation de Points Fixes D'Operateurs Monotones sur un Treillis, Analyse Semantique des Programmes*. Ph d. thesis, Université Scientifique et Médicale de Grenoble, Institut National Polytechnique de Grenoble, 1978.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *International Workshop of Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Grenoble, France, June 1990. Springer-Verlag.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. *The Tool KRONOS*. In *Proceedings of Hybrid Systems III*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the* 16th *IEEE Real-Time Systems Symposium (RTSS'95)*, pages 66–75, Pisa, Italy, December 1995. IEEE Computer Society Press.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. *Proceedings IEEE Real-Time Systems Symposium (RTSS '96)*, pages 73–81, December 1996.

[EMA+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *LNCS*, pages 346–360, Grenoble, France, 1997. Springer Verlag.

[For03] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*. The MPI Forum, 2003.

[GHS01] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. In *Computer Aided Verification*, pages 350–362, 2001.

[GM02] Michael Goldsmith and Jeremy Martin. Parallelization of FDR. In *Workshop on Parallel and Distributed Model Checking, afiliated to CONCUR 2002 (*13th *International Conference on Concurrency Theory)*, Brno, Czech Republic, August 2002.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina M. Smarandache. Parallel state space construction for model-checking. In Matthew B. Dwyer, editor, *Proc. of the 8th International SPIN Workshop*, pages 217–234, Toronto, Canada, 2001.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, 1988.

[HGGS02] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. *Formal Methods in System Design*, 21(2):317–338, November 2002.

[HKK02] Keijo Heljanko, Victor Khomenko, and Maciej Koutny. Parallelisation of the petri net unfolding algorithm. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, pages 371–385, 2002.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.

[Hol02] Gerard J. Holzmann. The logic of bugs. *SIGSOFT Softw. Eng. Notes*, 27(6):81–87, 2002.

[Hol03] Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[KK98] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical report, University of Minnesota, Department of Computer Science / US Army HPC Research Center. Minneapolis, USA., March 1998.

[KL98] K. Strehl and L. Thiele. Symbolic model checking using Interval Diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), 1998.

[Krc03] Pavel Krcal. Distributed explicit bounded LTL model checking. In Lubos Brim and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89 of *ENTCS*. Elsevier, 2003.

[LLPY03] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Compact data structures and state-space reduction for model-checking real-time systems. *Real-Time Syst.*, 25(2-3):255–275, 2003.

[LNZ05] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theor. Comput. Sci.*, 345(1):27–59, 2005.

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.

[NC97] David M. Nicol and Gianfranco Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, 1997.

[Oli94] Alfredo Olivero. *Modélisation et Analyse de Systèmes Temporisés et Hybrides*. Ph d. thesis, Institut National Polytechnique de Grenoble, 1994.

[Pav06] Esteban Pavese. *A new data structure based on BDDs for the model checking of timed systems*. Degree thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, April 2006. (`http://lafhis.dc.uba.ar/~epavese/tesis_pavese.ps.gz`).

[Pnu05] Amir Pnueli. Extracting controllers for timed automata. Technical report, Department of Computer Science, Weizmann Institute of Science, 2005.

[RSBSV96] R. Ranjan, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *International Conference on Computer Design*, pages 358–364, 1996.

[SBO02] Fernando Schapachnik, Víctor Braberman, and Alfredo Olivero. An Architecture-Centric Approach to the Development of a Distributed Model-Checker for Timed Automata. In *24th International Conference on Software Engineering*, pages 710–710. ACM Press, 2002.

[Sch02] Fernando Schapachnik. *Distributed and Parallel Verification of Real-Time Systems*. Degree thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, June 2002.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. In *Computer Aided Verification*, volume 1254 of *LNCS*, pages 256–278. Springer-Verlag, 1997.

[Shi96] Yuan Shi. Reevaluating Amdahl's Law and Gustafson's Law. Technical report, Computer and Information Sciences Department, Temple University, Philadelphia, USA, October 1996. http://knight.cis.temple.edu/ shi/docs/amdahl/amdahl.html.

[SKK00] Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. Technical report, University of Minnesota, Department of Computer Science / US Army HPC Research Center. Minneapolis, USA., May 2000.

[Tri98] S. Tripakis. *The analysis of timed systems in practice*. Ph d. thesis, Universite Joseph Fourier, Grenoble, Francia, December 1998.

[vD98] Stijn van Dongen. A new cluster algorithm for graphs. In *281*, page 42. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-3681, 31 1998.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Wan00] Farn Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *TACAS '00: Proceedings of the* 6th *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 157–171, London, UK, 2000. Springer-Verlag.

[Wan03a] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation:* 4th *International Conference, VMCAI*, volume 2575 of *LNCS*, pages 189–205. Springer-Verlag, 2003.

[Wan03b] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 189–205, London, UK, 2003. Springer-Verlag.

[Yov97] Sergio Yovine. Model checking timed automata. In G. Rozenberg and F. Vaandrager, editors, *Embedded Systems*, volume 1494 of *LNCS*. Springer-Verlag, 1997.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

[ZLZZ03] Jianhua Zhao, Xuandong Li, Tao Zheng, and Guoliang Zheng. Removing irrelevant atomic formulas for checking timed automata efficiently. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 34–45. Springer, 2003.

Appendix A

Case Studies

A.1. Railroad Crossing System

Fig. A.1 shows the three components of the *Railroad Crossing System* (*RCS* for short) inspired in the example presented in [ACD+92]. The controller registers the status of trains and actuates over the gate accordingly. In the case shown the controller can deal with up to two trains. They are synchronized by the labels: app_1, app_2, $exit_1$, $exit_2$, *lower* and *down*. When $\textsc{Train}_i$ approaches the crossing, it sends a signal app_i to Controller and enters the crossing at least 6, at most 10 time units later. When $\textsc{Train}_i$ leaves the crossing it sends a signal $exit_i$ to Controller within 15 time units after the signal app_i was sent. Controller sends a signal *lower* to the gate at most 1 time unit after the arrival of signal app_i (if the crossing is empty), and sends *raise* signal within 1 time unit receiving the signal $exit_i$ (if there is no other train detected). It takes 3 time units to raise and lower the Gate.

We will write $RCSn$ to mean a n-train version of the case study.

A.2. MinePump

In the *MinePump* example, a watchdog task periodically checks the availability of a water level sensor device by sending a request and extracting ACKs that were received and queued during the previous cycle by another sporadic task. When the watchdog finds the queue empty, it registers a fault condition in a shared memory which is periodically read, and forwarded to a remote console, by a proxy task. One property of interest is that the failure of high-low water sensor is always informed to the remote operator before a given deadline. Another property is that the values of CO and CH_4 logged were sampled no more than 100 ms away.

A.3. Conveyor Belt

By CBn, m we mean a *Conveyor Belt* modeled as a pipeline of stages where objects are fed at one extreme and consumed on the other (based on an example from [DY95]). Each stage and each flowing object is modeled by a timed automaton. The consumer has a latch that accumulates the signals produced by objects when they sojourn the last stage. We want to detect if there exists a case where an object flows in the pipeline for more than a given deadline. Being a scalable example, its size depends on two parameters: n the number of stages and m ($\leq n$) the number of objects.

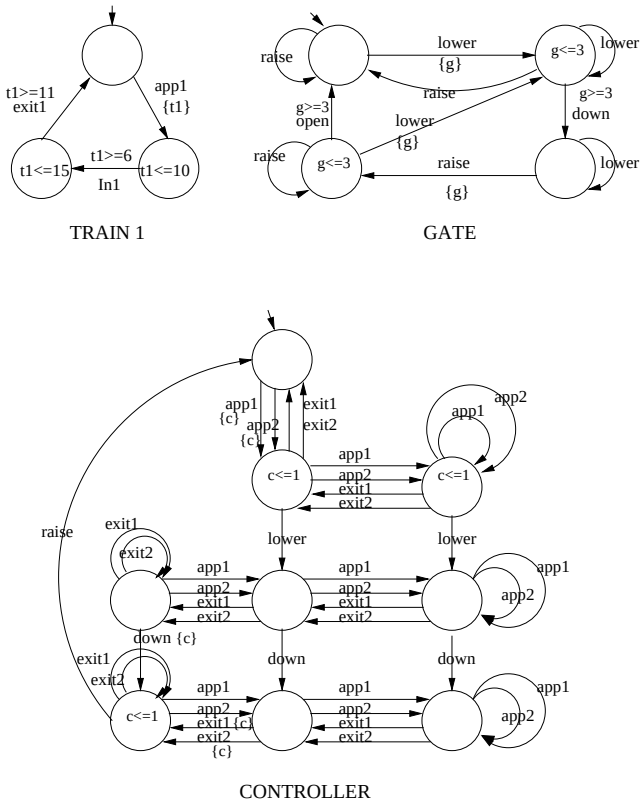


Figure A.1: The Railroad Crossing System, with two trains

A.4. FDDI

We write *FDDIn* to refer to a version of the well known FDDI token ring protocol [Tri98] as used in [BGO02], with n workstations.

For the backwards version, the case study was treated with Optikron [DY96] for inactive clock suppression.

A.5. RemoteSensing

The *RemoteSensing* case study presented in [ABKO04] can be instantiated in two versions, one testing for bounded response, *RS-BR*, and another for correlation, *RS-C*.

A.6. Pipe

Pipen is an end-to-end signal propagation in a pipe-line of sporadic processes that forward a signal emitted by a quasi periodic source, with n stages. There is a latch in the middle of each pair of processes.

List of Figures

List of Tables

List of Algorithms

Index