Accelerated Verification of Concurrent Systems

Accelerated Verification of Concurrent Systems

Maurice Laveaux

Accelerated Verification of Concurrent Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op dinsdag 22 november 2022 om 16:00 uur

door

Maurice Laveaux

geboren te Geleen

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr.ir. B. Koren
1e promotor:	dr.ir. T.A.C. Willemse
2e promotor:	prof.dr.ir. J.F. Groote
leden:	prof.dr. J.C. van de Pol (Aarhus University)
	prof.dr. A.W. Roscoe FREng (University of Oxford)
	prof.dr. B. Speckmann
adviseur:	dr. H.P. Hijma

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

This publication is part of the project Accelerated Verification and Verified Acceleration with project number 612.001.751 of the research programme TOP Grants which is (partly) financed by the Dutch Research Council (NWO).

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) IPA dissertation series 2022-10

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-5591-8

Printed by ADC Nederland

Cover design by Maurice Laveaux.

Preface

During my master's thesis I heard about an open PhD position related to GPU accelerated model checking algorithms during one of our weekly presentations and I expressed interest to my advisor Tim Willemse at the time. After successfully defending my master's thesis I got the opportunity to pursue a doctorate degree on this GPU related research project, with Tim as my main advisor. However, it will become apparent that none of the chapters in this dissertation are related to GPUs and they have only been used to render out the cover art. It turned out that on this project any research topic would be fine as long as it could lead to interesting results.

First of all, I would like to thank my promotors Tim Willemse and Jan Friso Groote for the opportunity to pursue a PhD. Most of all, I am grateful for the guidance and feedback provided by Tim Willemse over all these years, with whom I have contributed various papers. Furthermore, I would like to thank Jan Friso Groote for their technical insights and always being open for discussions as long as they are related to interesting research topics. Their passion for research is truly admirable, and the amount of time that Jan Friso dedicates to the mCRL2 toolset is astonishing. Over these years I made various contributions to this toolset, including the algorithms presented in this thesis.

I would like to thank my colleagues for the pleasant work environment that I have enjoyed since the start of my doctorate degree. First of all, this includes my (former) office mates Olav Bunte, Alexander Fedotov, Muhammed Osama Mahmoud, Thomas Neele, Flip van Spaendonck, and since recently Kevin Jilissen. Thomas was particularly helpful in getting new PhD candidates started by always being able to answer questions and integrating them into the group by organising various events. Furthermore, your ability to always stay organised and come up with clever suggestions is admirable. I already met Olav during my bachelor studies and over the years before and during my PhD we had a lot of interesting discussions and fun at many occasions.

Next, I would like to thank the other (former) PhD candidates of our group, including Mark Bouwman, Omar al Duhaiby, Rick Erkens, Tom Franken, Jan Martens, Anna Stragmalia, Mahmoud Talebi and Ferry Timmers for the pleasant (and occasionally overly complicated) discussions during lunch and coffee breaks. Rick's knowledge of

weight lifting and ability to explain that and many theoretical computer science topics clearly is a great trait to have for someone interested in teaching. We have also worked together on one paper, and our discussion channel has been a great source of enjoyment over the years. Ferry, Olav, Rick and I also enjoyed many (late) evening team building activities in the form of video games, which where particularly necessary during the lockdown. Thanks Anna for teaching me the importance of occasionally enjoying a good cappuccino, and Tom for showing me an endless amount of cat pictures.

Furthermore, I also thank (former) PhD candidates of other groups, including Rodin Aarssen, Kousar Aslam, Pavlo Burda, Priyanka Karkhanis, Sangeeth Kochanthara, Josh Mengerink, Mauricio Verano Merino, Jouke Stoel, Wessley Silva Torres, Lina Ochoa Venegas and Nan Yang for the fun interactions during various events; IPA events, barbecues, drinks and Christmas parties. This also goes out to the newer candidates including Nathan Cassee, Lars van den Haak, David Manrique Negrin, Hossain Muhammad Muctadir and Satrio Rukmono. I wish all of you the best in continuing your research and eventually finishing your dissertation where applicable.

Although our interactions where mostly confined to lunch breaks I would also like to thank the other (former) members of our cluster, Roel Bloo, Loek Cleophas, Herman Geuvers, Jeroen Keiren, Bas Luttik, Alexander Serebrenik, Erik de Vink, Wieger Wesselink, Anton Wijs and Hans Zantema for the pleasant work environment and discussions. Wieger's contribution to the mCRL2 toolset cannot be overstated. His ability to write down pseudocode and then implement it an elegant way that can withstand the test of time is something that many programmers can learn a great deal from, including myself. Thanks Loek and the PhD council for all the great IPA events.

Furthermore, I would like to thank the support staff consisting of Agnes van den Reek and Margje Mommers-Lenders for help with organisational tasks.

I am grateful for the feedback and suggestions provided by the members of the reading committee, consisting of Pieter Hijma, Jaco van de Pol, Bill Roscoe and Bettina Speckmann.

I would like my parents for supporting me during my studies as that eventually allowed me to pursue a PhD. Over all these years you have fostered my curiosity to figure out how things work, starting with mechanical objects and then later on more abstract subjects. You have always allowed me to work on my own things and this has helped a great deal in being able to be self-sufficient. To all my friends that I met before and during my PhD, thanks for the good times.

Finally, since this dissertation is related to scalability I conclude with one final statement.

The factory must grow.

Maurice Laveaux

Contents

Chapter 1

Introduction

Computers are ubiquitous in our daily lives, ranging from the small microchip embedded in your debit card to the large data centers that form the foundation of the internet. Although all of these computers are immensely useful, many people are familiar with the fact that computers do not always work properly. For example, 80 percent of users notice bugs in mobile apps [127]. These issues can be (small) inconveniences or application crashes that can result in the loss of work. However, in a safety critical environment, computer failure can lead to fatal accidents. Recently, an issue in the safety alert system of the Boeing 737 Max resulted in two fatal crashes [73]. Other prominent examples are security exploits that (can) result in sensitive data being leaked into the public. A recent example being the ProxyLogon exploit targeting Microsoft Exchange servers [101]. In these cases software problems cause the loss of resources, privacy and even lives.

The essential observation about these issues is that the *actual* behaviour of the computer is not the behaviour *intended* by its designers. The behaviour of a computer is governed by two parts: the hardware and the software. The *hardware* is the collection of physical components of the computer. The *software* consists of the instructions that are executed by the hardware. In 1994, a design flaw in the floating point unit of certain Intel processors caused inaccurate results, which led to a large recall of sold processors. This problem, known as the infamous Pentium FDIV bug, has resulted in a significant increase in the use of rigorous design techniques such as formal verification at Intel [108]. These efforts among others have significantly reduced the presence of incorrect behaviour in common general purpose processors. Therefore, most faults these days are the result of mistakes in the software.

The need for ensuring the correctness of software, where *correct* means that the actual behaviour matches the intended behaviour, was quickly identified when

computers first appeared. In the early days a computer was modelled as a device that computes the output corresponding to the input values. This type of mapping from inputs to outputs is called a *function*. For functions and procedures written down as pseudocode, which is a formal way to describe programs, we are able to verify properties or invariants of them either by hand or using computer assistance. Although this is sufficient to prove the behavioural correctness for sequential programs it is missing the *interaction* of different components. Furthermore, the sheer complexity of modern software systems makes applying such an approach to these systems difficult due to the amount of insight required in both the approach and the system.

There are many complementary techniques to ensure the correctness of software. For the source code itself there are development processes to improve the quality of the code such as rigorous programming practices, extensive code reviewing processes and documenting the source code. Furthermore, the use of high-level languages that employ expressive type systems and have rules that can be verified by the compiler can help in avoiding bugs [49]. Sometimes specific programming language features can be a source of bugs, for example the widespread usage of null pointers [79].

For verifying the behaviour of software directly there are several techniques that can operate while the software is being executed. For example, we can introduce runtime assertions to avoid typical sources of bugs such as out-of-bounds errors and to verify other so-called *invariants* during execution. Furthermore, tests can be used to verify the results in specific situations. Finally, a more rigorous approach is model based testing, which directly compares the behaviour of the executed program directly with the intended behaviour. These methods are quite effective at detecting issues, but their disadvantage is that they only guarantee correctness of the software in the behaviour encountered during execution, which is inherently incomplete.

Formal verification is a technique that aims to verify the *complete* correctness of behaviour by modelling behaviour and verifying it directly. This has the advantage that we can guarantee the correctness of all the behaviour that was modelled. Furthermore, another advantage is that (part of) the program code can often be directly generated from the model. There are several (complementary) techniques to perform formal verification each with their advantages and disadvantages. For example, program verification tools such as Dafny [96], KeY [35] and VerCors [10], theorem provers such as Coq [133], Isabelle [107] and PVS [109], and model checking tools such as CADP [50], FDR [53], LTSmin [12] and mCRL2 [23]. Scalability of these tools is one of the main challenges for applying formal verification in practice. In this dissertation we focus on both theoretical and practical scalability improvements of several techniques employed in model checking of concurrent systems specifically. Hence the title accelerated verification of concurrent systems.

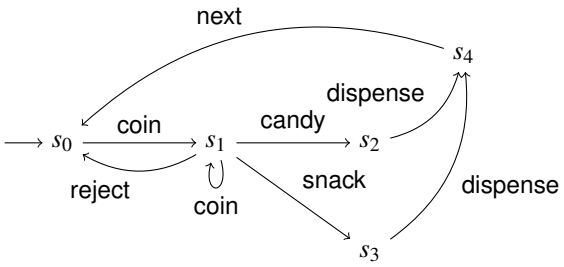


Figure 1.1: Behaviour of a vending machine.

1.1 Behavioural Equivalences

To describe the behaviour of interacting components we need a language that can express more than only functions. This observation gave rise to the development of *concurrency theory* where interaction between components is often modelled as messages being passed from one to another or data that is being shared. As part of this theory formal languages called process algebras have been developed to effectively describe both the sequential and interacting behaviour of processes. We are not going to delve into the details of process algebra, but we will explain the notion of behaviour in more concrete terms.

Let's consider the behaviour of a vending machine. A vending machine is in simple terms a (physical) machine that accepts money in some form and returns a product after a choice has been made by the user. This kind of behaviour can be *modelled* as an edge-labelled graph as is shown in Figure 1.1. Here, we see a number of *states* with names $s_0, s_1, \ldots$ and arrows indicating *transitions* from one state to another. The state s_0 has an incoming edge without a starting state to indicate that it is the *initial* state. We refer to these kind of graphs as *state spaces*. Every transition is labelled with a corresponding action that takes place. The interpretation of this graph is as follows. Whenever the machine is in (for example) state s_0, it can accept a coin from the user and it ends up in state s_1. The actions are essentially the externally observable behaviour of this machine, whereas the exact state that the machine is in cannot be observed directly.

If we are satisfied with the high-level behaviour of our system we could view this as the *specification* of the intended behaviour. However, this model leaves out a lot of details about the inner workings of a vending machine such as controlling the actuators that are used to actually dispense the product. Therefore, the actual system designers could also define an *implementation* model that contains the more refined behaviour of the system. There are techniques available to automatically verify whether these

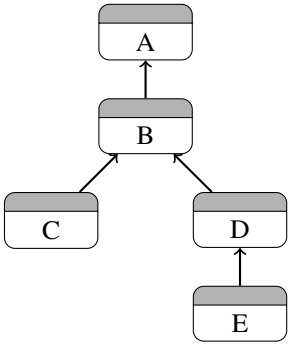


Figure 1.2: A refinement based software development approach.

specification and implementation models are *behaviourally equivalent* with respect to some notion of equivalence. Intuitively, we can for example observe in Figure 1.1 that after entering either state s_2 or s_3 the subsequent behaviour is exactly the same, so we would also consider the states to be indistinguishable. This could be used to verify that the actual behaviour matches the intended behaviour.

Behavioural equivalence is often considered too strong as a requirement, because we could view many (different) implementations as correct with respect to a given specification. Therefore, there are also the *weaker* notions of certain implementation-specification *refinement* relations, such as the weak trace and stable failure relations presented in [121]. These notions have been used successfully in industrial settings as a refinement based software development, for example in [8]. The idea is that we can replace implementations by specifications, which are typically much smaller, during the verification process.

Consider Figure 1.2 where A, B, C, D and E are components of some software system. Every component has an implementation indicated by the white part and a specification indicated by the grey part that is considered to be the *interface* of that implementation. The arrows indicate that components are *used by* other components, for example component B is used by component A. The main benefit of a refinement based software development approach is that it can be applied *compositionally*. This means that to verify the correctness of component A with respect to its specification we only need to consider the implementation of A and the interface of B. Thus we can ignore the fact that B uses components C, D and even indirectly E.

Since the state spaces of these components can be quite large it is important to use efficient algorithms to perform the refinement checking automatically. In [137] three variations of an algorithm based on so-called *antichains* was presented to check

(weak) trace, stable failures and failures-divergences refinement. Closer inspection of the algorithm and surrounding definitions revealed several issues pertaining to the correctness of these algorithms. Furthermore, our first implementation of this technique in our open source mCRL2 toolset yielded disappointing performance [23]. Therefore, the first research question we pose is the following.

RQ1 Can we resolve the issues pertaining to the correctness and efficiency of the antichain-based refinement algorithm presented in [137]?

In Chapter 2 we answer this question positively by slightly adapting the algorithms to drastically improve their performance. Furthermore, we have resolved several flaws in the invariants and definitions used in the proof of correctness. The result of this work is a collection of correct and efficient antichain-based refinement algorithm that are implemented in the mCRL2 toolset, and are currently used as the backbone in the commercial model driven engineering toolset Dezyne; see also [8].

1.2 Compositional Minimisation

In the previous section we assumed that the behaviour of a system was modelled as a graph structure. However, in practice the behaviour is often described in a high-level description language such as the aforementioned process algebras. These process algebras have constructs to represent various kinds of behaviour compactly including the interaction of multiple processes. These interactions are often between multiple processes that are executed *concurrently* and pass messages carrying data to each other. The underlying behaviour is then obtained from this high-level description as a state space by means of *state space exploration*. This concurrency often results in a lot of interleaving behaviour that causes the complete system behaviour to become very large. This phenomenon is one reason for the so-called *state space explosion problem* and a large part of research in the area of formal verification is focussed on this problem. It is however important to note that this complex behaviour is in principle part of the system.

There are various techniques described in the literature to deal with the state space explosion problem and one of them is *compositional minimisation*. For a detailed overview of this technique see [51]. Consider the Figure 1.3 where P_1, P_2 and P_3 are the state spaces of three processes. The basic idea of compositional minimisation is to minimise the state space of every process by itself with respect to some equivalence relation, which is a minimal state space representing *equivalent* behaviour, before composing these minimised state spaces again to obtained the complete state space. The node $P_1 \parallel P_2$ is the so called *parallel composition* of P_1 and

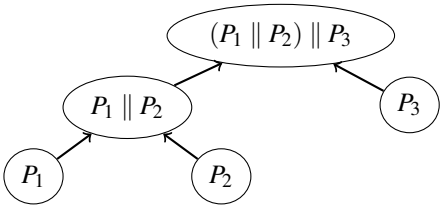


Figure 1.3: Compositional minimisation of state spaces.

P_2, which is the behaviour of running these two processes in parallel. Next, $P_1 \parallel P_2$ can be minimised again before being put in parallel with component P_3. The reason why this is useful is that for smaller state spaces the amount of possible interleaving is also reduced. Applying this technique in practice is quite a bit more tricky than this simple explanation since we do not only have processes that are executed concurrently, but most often the processes also communicate or *synchronise* information with each other. This means that the behaviour of one process can be limited by other processes and the state space of a single component in isolation can be larger than the complete system, or even infinitely large.

In the mCRL2 toolset a transformation is applied from the high-level description into an equivalent non-deterministic *monolithic* process before further processing. This monolithic process is a process that only uses a few typical constructs from process algebra and in particular no longer contains parallel composition. The previously described compositional approach relies on the parallel composition in the original process descriptions and is therefore no longer directly applicable. We could consider adapting this transformation to result in a number of parallel components. However, in practice it is often quite useful to apply static analysis techniques on the high-level description to obtain (significant) reductions in the state space of the system. These static analysis techniques are much easier to implement and prove correct for monolithic processes and also the implementation of state space exploration is greatly simplified, which are good reasons to keep the transformation as it is. We can also observe that in principle the information necessary to apply compositional minimisation is still contained in this monolithic processes. Therefore, the second research question that we pose is the following.

RQ2 Can compositional minimisation be applied to a monolithic process by decomposing it into multiple monolithic processes?

In Chapter 3 we present a technique that can decompose a monolithic process

of a certain general structure into multiple smaller components using only operators available in a process algebra with so-called multi-actions. Note that the mCRL2 language [63] is an example of a process algebra with multi-actions. These multi-actions are not necessarily widely available in other process algebras, but do present a nice way to represent actions that occur simultaneously. Furthermore, we demonstrate using practical examples that the components resulting from our decomposition can be used for compositional minimisation. Finally, we show on several practical examples that the resulting decomposition can be explored more efficiently than the monolithic state space exploration.

1.3 Symbolic Model Checking

In Section 1.1 we considered verifying behaviour by checking whether two models were in some sense equivalent or a refinement of each other. Although this is a useful approach for formal verification there might also be specific *requirements* that (the behaviour of) the system should exhibit that are not necessarily preserved by the equivalence or refinement relation. Therefore, another complementary approach is to specify requirements directly in a formal language, for which examples are the modal μ-calculus [63] and LTL [113].

Consider the state space presented in Figure 1.1 again. A typical requirement is that the system does not *deadlock*, meaning that there is no state without any outgoing transitions. Since the modal μ-calculus reasons about observable behaviour, *i.e.*, the actions that take place, we specify this as: after any path of transitions we reach a state which has at least one outgoing transition. Formally, this would be stated as a sentence in the modal μ-calculus shown below, for which the details are currently not important. This property clearly holds for the given state space, and this can also checked automatically.

$$[\mathsf{true}*]\langle\mathsf{true}\rangle\mathsf{true}$$

A more specific requirement might be that after inserting a *coin* within a finite number of steps either *candy* or *snack* is dispensed from the machine. However, the infinite path consisting of only coin actions by continuously looping in state s_1 shows that this requirement is not satisfied by the given behaviour. This means that we should either adapt the specification or refine the requirement. On the other hand, if we just require that there is an infinite path that consists of alternating coin and snack actions then that requirement does hold.

The process of automatically verifying whether requirements hold for a process described by a high-level language is referred to as *model checking*. One approach is to encode the model checking question into a so-called *parity game* [140]. This is a game that is played on a finite directed graph, such as the example shown in Figure 1.4.

Every vertex of the graph is owned by either player even, which has a diamond shape ◇, or odd, which has a box shape □. Furthermore, every vertex is labelled by a number that indicates their priority. The owner and priority of every vertex are determined by the formula from which the parity game is derived.

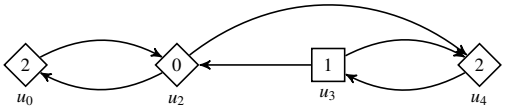


Figure 1.4: An example parity game

Now that we have introduced the players and the arena on which the game is played it is time to explain how the game is played. First of all, a token is placed on any of the vertices of the graph. Next, the owner of the vertex can move the token along any of their outgoing edges. During this so-called *play* we keep track of the sequence of priorities visited by the token and this process essentially continues forever. The *winning condition* is that the least priority occurring infinitely often has your parity, so if it is even then player even wins and otherwise player odd wins. For parity games it is known that the winning partition, *i.e.*, the disjoint sets of vertices won by player even and odd respectively, can be computed [140].

These parity games encode information about both the behaviour of the system and the requirement that is being checked. Therefore, they also suffer from the same kind of *state space explosion* as the underlying behaviour. This means that it makes sense to study compact representations for the vertices and edges of the parity game that also allow for efficient operations needed for computations. One data structure that *can* be exponentially more compact than a naive representation to represent a set of tuples is the *multi-valued decision diagram* [102]. In Figure 1.5 we show a number of tuples and their compact representation using a multi-valued decision diagram.

Recently there has been an effort to extend the capabilities of our mCRL2 toolset with techniques that utilise symbolic representations of the underlying parity games, which is based on [89]. Since our toolset has had extensive development on the explicit counterpart there were many techniques in the explicit setting that could be applied onto the symbolic techniques. The question was which of these techniques can also be adapted to symbolic representations, and one of these techniques was on-the-fly solving of parity games. Similarly to how a labelled transition is derived from a high-level description there is a process to derive a parity game from a syntactic representation. The essential idea of on-the-fly solving is that during this exploration process we can solve the parity game with *incomplete* information that has been explored so far to *eagerly* try to find the winner for the vertex corresponding to the

⟨2 “a” true⟩
⟨2 “b” true⟩
⟨2 “b” false⟩
⟨7 “a” true⟩
⟨7 “a” false⟩

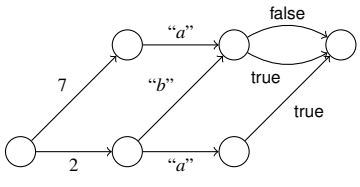


Figure 1.5: A multi-valued decision diagram (right) for the set of tuples on the left.

initial state. This is called eager since it might not yet be possible to determine the winner for the initial vertex at this time.

RQ3 Can on-the-fly solving be applied effectively to symbolic parity games?

In Chapter 4 we present a notion of *incomplete* parity games to study the notion of on-the-fly solving for parity games. We show a method to obtain two winning sets and an *undecided* set for the incomplete parity games based on the notion of *safe* vertices. Furthermore, we show that this method is *optimal* in the sense that the undecided set is as small as possible w.r.t. the set of all possible extensions of the incomplete parity game. Since parity game solvers have (worst-case) quasi-polynomial, and more practical exponential, running times we have also studied several *partial* solvers with polynomial running time. These partial solvers might only determine winners for part of the game, but have better performance. For three partial solvers based on solitaire cycles, winning cycles and fatal attractors we have introduced alternatives that operate directly on the incomplete parity games without computing the safe vertices as a preprocessing step. Finally, we have implemented our technique and performed an experimental evaluation to show that this technique can also work well in practice.

1.4 Thread-safe Term Library

A term is a fundamental concept in mathematics and therefore an equally important common data structure in computing. Many concepts are terms, such as programs, specifications and formulas, see for example Figure 1.6. Many operations in computing are transformations of these terms, one example being the compilation, *i.e.*, the transformation from a program into executable code. In computer science a term is a far more commonly used concept than structures such as arrays, lists or matrices.

This makes it remarkable that terms are not a standard data structure in all common programming languages. Many tools therefore use some kind of internal data structure to represent and operate on terms and our mCRL2 toolset is no exception.

These so-called *first-order* terms have a very simple structure that makes them so versatile. First of all, there is a set of *function symbols* that are symbols which can represent mathematical operations such as $+$, $*$ or any functions such as f or g. Every function symbol has an associated *arity*. For example addition $(+)$ must be applied to two numbers, so its arity is two since its applied to two arguments. Next, we can construct other terms by applying a function symbol to its appropriate amount of terms. For example $+(3,6)$ is a term provided that 3 and 6 are two *constants*, which are function symbols with arity zero. For binary operations we typically write $3+6$ for readability, but in general we prefer the notation $f(3,6)$. As seen before $3+4*2$ is also a term.

When terms are used inside software there needs to be a way to effectively create, inspect and manipulate terms inside the program. For this purpose it makes sense to have a general purpose term *library*, which is an implementation that has a well-defined interface by which it can be invoked.

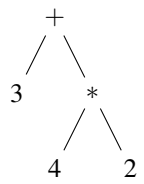


Figure 1.6: The tree representation of the term $3+4*2$.

Therefore, we first introduce some of the desired features that our term library should have. Since there can be many terms in use during computation we employ so-called *maximal sharing* to ensure that every term only occurs once in memory, *i.e.*, in the term $f(g,g)$ there is only one instance of g no matter how complex this term is. This also has the advantage that terms can be easily compared, since two terms share the same address iff they are the same. Furthermore, we also have to take care that terms can go out of scope and must be cleaned up from memory again, which is called *garbage collection* since unreachable terms are essentially garbage.

With a steadily increasing number of computational cores in computers, it is also desirable to have a term library that can be used by multiple threads concurrently. This would allow for example *term rewriting*, which is a fundamental operation of the mCRL2 toolset, to also be applied in parallel since it creates many intermediate terms during its computation. This would in turn mean that state space generation can be parallelised, which would reduce the time it takes to perform model checking. However, the maximal sharing of terms makes this quite non-trivial and previous attempts have all failed at achieving desirable performance. Therefore, we propose the following research question.

RQ4 Can a *thread-safe* term library to store and manipulate terms be developed

that achieves *linear speedup* compared to an efficient sequential implementation?

In Chapter 5 we present a formal specification of a *thread-safe* term library, *i.e.*, a term library that can be used by multiple threads in parallel. We identify the need for a *readers-writer* lock implementation, *i.e.*, multiple threads can share a resource or a writer can obtain *exclusive* access, that is efficient for read heavy workloads. To this end we develop a new *busy-forbidden* protocol that only writes one atomic variable that is typically in the local cache and reads another that is rarely updated to obtain read access. For this new protocol we develop a formal specification and model check the corresponding implementation model with key properties for finite instances up to seven threads. Furthermore, we also model check the specification of the complete term library for several finite instances. Finally, we implement a prototype of this term library in the mCRL2 toolset. Although a perfect linear speedup has not been achieved, we can demonstrate that that the state space generation scales well for multiple threads, achieving a 12 times speedup for 16 threads.

1.5 Contributions

The work presented in this dissertation has led to several publications. Chapter 2 is based on a journal article published in LMCS [93]. The next chapter (Chapter 3) is based on the journal version of [95] that is under submission at JLAMP. Chapter 4 is based on a conference article presented at TACAS [94]. Finally, Chapter 5 is based on a conference article that has been accepted for publication at ISoLA, and was also fundamental to another publication [66]. In this last work my contribution has been mostly the benchmarking and implementation of the prototype, but have also been involved in discussions surrounding the formulas that have been verified. Additional work that has been carried out, but does not appear in this dissertation, on non-linear pattern matching automata is published at LMCS [45]. There, my contributions were the definition and proofs for the so-called *consistency automata* that are used to deal with the equality constraints. Finally, I have also been involved in the implementation of an extension to the decomposition method described in Chapter 3. This extension to our work has been accepted for publication at FACS.

Chapter 2

Antichain Algorithms for Refinement Checking

Refinement is often an integral part of a mature engineering methodology for designing a (software) system in a stepwise manner. It allows one to start from a high-level specification that describes the permitted and desired behaviours of a system and arrive at a detailed implementation that behaves according to this specification. While in many settings, refinement is often used rather informally, it forms the mathematical cornerstone in the theoretical development of the process algebra CSP (Communicating Sequential Processes) by Hoare [78, 121, 122].

This formal view on refinement—as a mathematical relation between a specification and its implementation—has been used successfully in industrial settings [61, 52], and it has been incorporated in commercial Formal Model-Driven Engineering tools such as *Dezyne* [8]. In such settings there are a variety of refinement relations, each with their own properties. In particular, each notion of refinement offers specific guarantees on the (types of) behavioural properties of the specification that carry over to correct implementations. For instance, *trace refinement* [122] only preserves safety properties. The—arguably—most prominent refinement relations for the theory of CSP are the *stable failures refinement* [7, 122] and *failures-divergences refinement* [122]. All three refinement relations are implemented in the FDR [53, 54] tool for specifying and analysing CSP processes.

Checking for trace refinement, stable failures refinement and failures-divergences refinement are computationally hard problems; deciding whether there is a refinement relation between an implementation and a specification, both represented by CSP processes or labelled transition systems, is PSPACE-hard [87]. In practice, however,

tools such as FDR are able to work with quite large state spaces. The basic algorithm for deciding a trace refinement, stable failures refinement or a failures-divergences refinement between implementation and specification relies on a *normalisation* of the specification. This normalisation is achieved by a subset construction that is used to obtain a deterministic transition system which represents the specification.

As observed in [137] and inspired by successes reported, *e.g.*, in [1, 42, 139], *antichain* techniques can be exploited to improve on the performance of refinement checking algorithms. Unfortunately, a closer inspection of the results and algorithms in [137] reveals several issues. First, the definitions of stable failures refinement and failures-divergences refinement used in [137] do not match the definitions of [7, 122], nor do they seem to match known relations from the literature [55].

Second, as we demonstrate in Example 2.3.3 in this chapter, the results [137, Theorems 2 and 3] claiming correctness of their algorithms for deciding both non-standard refinement relations are incorrect. We do note that their algorithm for checking trace refinement is correct, and their algorithm for checking stable failures refinement correctly decides the refinement relation defined by [7, 122].

Third, unlike claimed by the authors, the algorithms of [137] violate the antichain property as we demonstrate in Example 2.3.5. Fourth, their algorithms suffer from severely degraded performance due to sub-optimal decisions made when designing the algorithms, leading to an overhead of a factor $|Act| \cdot |S|$, where Act is the set of actions and S the set of states of the implementation, as we show in Example 2.3.4. This factor is even greater, *viz.* $|Act|^{|S|}$, when using a FIFO (first in, first out) queue to realise a breadth-first search strategy instead of the stack used for the depth-first search. Note that there are compelling reasons for using a breadth-first strategy [121]; *e.g.*, the conciseness of counterexamples to refinement.

Our contributions are the following. Apart from pointing out the issues in [137], we propose new antichain-based algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement and we prove their correctness. We compare the performance of the trace refinement algorithm and the stable failures refinement algorithm of [137] to ours. Due to the flaw in their algorithm for deciding failures-divergences refinement, a comparison of this refinement relation makes little sense. Our results indicate a small improvement in run time performance for practical models when using depth-first search, whereas our experiments using breadth-first search illustrate that decision problems, intractable using the algorithm of [137], generally become quite easy using our algorithm. Finally, we show that divergence-preserving branching bisimulation [60, 56] minimisation preserves the desired refinement checking relations and that applying this minimisation as a preprocessing step can yield significant run time improvements.

Outline In Section 2.1 the preliminaries of labelled transition systems and the refinement relations are defined. In Section 2.2 a general procedure for checking refinement relations is described. In Section 2.3 the antichain-based algorithms of [137] are presented and their issues are described in detail. In Section 2.4 the improved antichain algorithms are presented and their correctness is shown. Finally, in Section 2.5 an experimental evaluation is conducted to show the effectiveness of these changes in practice, followed by the evaluation of applying divergence-preserving branching bisimulation minimisation as a preprocessing step.

2.1 Preliminaries

In this section the preliminaries of labelled transition systems and the considered refinement relations are introduced. We follow the standard conventions, notation and definitions of [19, 122, 59].

2.1.1 Labelled Transition Systems

Let Act be a finite set of actions that does not contain the constant τ, which models *internal* actions, and let Act_τ be equal to $Act \cup \{\tau\}$.

Definition 2.1.1. A labelled transition system $\mathscr{L}$ is a tuple $(S, \iota, \rightarrow)$ where S is a set of states; $\iota \in S$ is an initial state and $\rightarrow \subseteq S \times Act_\tau \times S$ is a labelled transition relation.

We depict labelled transition systems as edge-labelled directed graphs, where vertices represent states and the labelled edges between vertices represent the transitions. An incoming arrow with no starting state and no action indicates the initial state. We use the initial state to refer to a depicted LTS.

For the remainder of this section, we assume that $\mathscr{L} = (S, \iota, \rightarrow)$ is an arbitrary LTS. We adopt the following conventions and notation. Typically, we use symbols s, t, u to denote states, U, V to denote sets of states and a to denote actions. A transition $(s, a, t) \in \rightarrow$ is also written as $s \xrightarrow{a} t$. The set of *enabled* actions of state s is defined as $\mathsf{enabled}(s) = \{a \in Act_\tau \mid \exists t \in S : s \xrightarrow{a} t\}$.

A sequence is denoted by concatenation, *i.e.*, $a_0\, a_1 \cdots a_{n-1}$ where $a_i \in Act_\tau$ for all $0 \leq i < n$ is a sequence of actions and $Act_\tau{}^*$ indicates the set of all finite sequences of actions. We use σ and ρ to denote a sequence of actions, where ρ typically does not contain τ. The *length* of a sequence, denoted as $|a_0\, a_1 \cdots a_{n-1}|$, is equal to n. Finally, we say that any sequence $a_0\, a_1 \cdots a_k$ such that $k \leq n-1$ is a *prefix* of a sequence $a_0\, a_1 \cdots a_{n-1}$. A prefix is *strict* whenever its length is strictly smaller than that of the sequence itself.

The transition relation of an LTS is generalised to sequences of actions as follows: $s \xrightarrow{\varepsilon}\!\!\!\!\rightarrow t$ holds iff $s = t$, and $s \xrightarrow{\sigma a}\!\!\!\!\rightarrow t$ holds iff there is a state u such that $s \xrightarrow{\sigma}\!\!\!\!\rightarrow u$ and $u \xrightarrow{a} t$. The *weak transition* relation $\Longrightarrow \subseteq S \times Act^* \times S$ is the smallest relation satisfying:

- $s \overset{\varepsilon}{\Longrightarrow} s$, and
- $s \overset{\varepsilon}{\Longrightarrow} t$ if $s \xrightarrow{\tau} t$, and
- $s \overset{a}{\Longrightarrow} t$ if $s \xrightarrow{a} t$ for $a \in Act$, and
- $s \overset{\rho\,\sigma}{\Longrightarrow} t$ if there is a state u such that $s \overset{\rho}{\Longrightarrow} u$ and $u \overset{\sigma}{\Longrightarrow} t$.

Definition 2.1.2. Traces, weak traces and reachable states are defined as follows:

- The *traces* starting in state s are defined as $\mathsf{traces}(s) = \{\sigma \in Act_\tau{}^* \mid \exists t \in S : s \xrightarrow{\sigma}\!\!\!\!\rightarrow t\}$. We define $\mathsf{traces}(\mathscr{L})$ to be $\mathsf{traces}(\iota)$.
- The *weak traces* starting in state s are defined as $\mathsf{weaktraces}(s) = \{\rho \in Act^* \mid \exists t \in S : s \overset{\rho}{\Longrightarrow} t\}$. We define $\mathsf{weaktraces}(\mathscr{L})$ to be $\mathsf{weaktraces}(\iota)$.
- the set of states, *reachable* from s is defined as $\mathsf{reachable}(s) = \{t \in S \mid \exists \sigma \in Act_\tau{}^* : s \xrightarrow{\sigma}\!\!\!\!\rightarrow t\}$. We define $\mathsf{reachable}(\mathscr{L})$ to be $\mathsf{reachable}(\iota)$.

Definition 2.1.3. Labelled transition system $\mathscr{L}$ is:

- *deterministic* if and only if for all states s,t,u and actions $a \in Act_\tau$ if there are transitions $s \xrightarrow{a} t$ and $s \xrightarrow{a} u$ then $t = u$.
- *concrete* if it does not contain transitions labelled with τ, *i.e.*, for all states s it holds that $\tau \notin \mathsf{enabled}(s)$.
- *universal* if and only if for all states s it holds that $\mathsf{enabled}(s) = Act$.

Lemma 2.1.4. Let $\mathscr{L}$ be a deterministic LTS. For all sequences $\sigma \in Act_\tau{}^*$ and states s,t,u, if $s \xrightarrow{\sigma}\!\!\!\!\rightarrow t$ and $s \xrightarrow{\sigma}\!\!\!\!\rightarrow u$ then $t = u$.

The models underlying the CSP process algebra [78, 122] build on observations of *weak traces*, *failures* and *divergences*. A weak trace observation records the visible actions that occur when performing an experiment on the system. A failure is a combination of a set of actions that a system observably refuses and a weak trace experiment on the system that leads to the observation of the refusals. A refusal can only be observed when the system has *stabilised*, meaning that it can no longer perform internal behaviour. A *divergence* can be understood as the potential inability of the system to stabilise, which can happen when the system engages in an infinite sequence of τ-actions after performing an experiment on the system.

Definition 2.1.5 (Refusals)**.** A state s is *stable*, denoted by $\mathsf{stable}(s)$, if and only if $\tau \notin \mathsf{enabled}(s)$. For a stable state s, the *refusals* of s are defined as $\mathsf{refusals}(s) = \mathscr{P}(Act \setminus \mathsf{enabled}(s))$. For a set of states $U \subseteq S$ its refusals are defined as $\mathsf{refusals}(U) = \{X \subseteq Act \mid \exists s \in U : \mathsf{stable}(s) \wedge X \in \mathsf{refusals}(s)\}$.

Formally, a state s is *diverging*, denoted by the predicate $s\Uparrow$, if and only if there is an infinite sequence of states $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots$. For a set of states U, we write $U\Uparrow$, iff $s\Uparrow$ for some state $s \in U$.

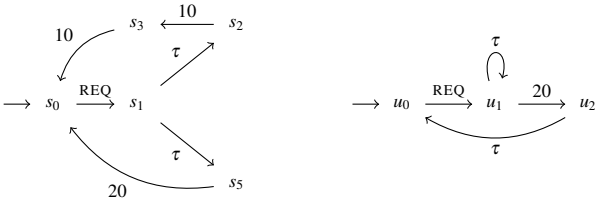
Definition 2.1.6 (Divergences)**.** The *divergences* of a state s are defined as the function $\mathsf{divergences}(s) = \{\rho\,\sigma \in Act^* \mid \exists t \in S : (s \stackrel{\rho}{\Longrightarrow} t \wedge t\Uparrow)\}$. We define $\mathsf{divergences}(\mathscr{L}) = \mathsf{divergences}(\iota)$.

Observe that a divergence is any weak trace that has a prefix ρ which can reach a diverging state. This is based on the assumption that divergences lead to *chaos*. In theories such as CSP, in which divergences are considered chaotic, chaos obscures all information about the behaviours involving a diverging state; we refer to this as obscuring *post-divergences details*.

Definition 2.1.7 (Stable failures)**.** The set of all *stable failures* of a state s is defined as $\mathsf{failures}(s) = \{(\rho, X) \in Act^* \times \mathscr{P}(Act) \mid \exists t \in S : (s \stackrel{\rho}{\Longrightarrow} t \wedge \mathsf{stable}(t) \wedge X \in \mathsf{refusals}(t))\}$. The set of failures with post-divergences details *obscured* is defined as $\mathsf{failures}_\perp(s) = \mathsf{failures}(s) \cup \{(\rho, X) \in Act^* \times \mathscr{P}(Act) \mid \rho \in \mathsf{divergences}(s)\}$.

We illustrate these concepts by means of an example.

Example 2.1.8. Consider the LTSs s_0 and u_0 depicted below.



We observe that states s_0, s_2, s_3, s_5 and u_0 are stable. For each of these states we can determine their refusals, *e.g.*, state s_0 has the refusals $\{\emptyset, \{10\}, \{20\}, \{10, 20\}\}$ as given by $\mathsf{refusals}(s_0)$. Furthermore, we observe that REQ 20 is a weak trace of s_0 to itself. Consequently, it follows that for example the pairs $(\text{REQ}\,20, \{10\})$ and $(\text{REQ}\,20, \{10, 20\})$ are failures of s_0. None of the states in s_0 diverge and as such the corresponding set of divergences are empty and both notions of failures coincide. However, for state u_1 we can see that $u_1\Uparrow$ holds and therefore REQ, but also REQ 10 is a possible divergence of u_0, *i.e.*, $\text{REQ}\,10 \in \mathsf{divergences}(u_0)$. This also means that $(\text{REQ}\,10, \{10\}) \in \mathsf{failures}_\perp(u_0)$ is a failure of u_0 with post-divergences details obscured. □

The three models of CSP building on the different powers of observation are the *weak trace* model, the *stable failures* model and the *failures-divergences* model. The refinement relations, induced by these models, are called *trace refinement*, *stable failures refinement* and *failures-divergences refinement* respectively.

Definition 2.1.9 (Refinement)**.** Let $\mathscr{L}_1$ and $\mathscr{L}_2$ be two LTSs.

- $\mathscr{L}_1$ is refined by $\mathscr{L}_2$ in trace semantics, denoted by $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$, if and only if $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$.
- $\mathscr{L}_1$ is refined by $\mathscr{L}_2$ in stable failures semantics, denoted by $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$, if and only if $\mathsf{failures}(\mathscr{L}_2) \subseteq \mathsf{failures}(\mathscr{L}_1)$ and $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$.
- $\mathscr{L}_1$ is refined by $\mathscr{L}_2$ in failures-divergences semantics, denoted by $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$, if and only if both $\mathsf{failures}_\perp(\mathscr{L}_2) \subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$ and $\mathsf{divergences}(\mathscr{L}_2) \subseteq \mathsf{divergences}(\mathscr{L}_1)$.

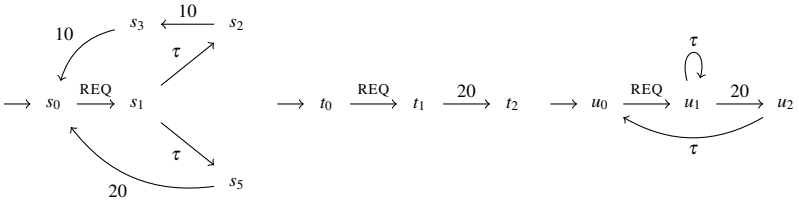
The LTS that is typically refined is referred to as the *specification*, whereas the LTS that refines the specification is referred to as the *implementation*.

Remark 2.1.10. We observe that each refinement relation between LTSs is inverted with respect to the subset relation in its corresponding definition. However, in the setting of CSP where these refinement relations are fundamental and have been extensively studied [19, 122, 59], refinement is viewed as an ordering between processes where the process that does not restrict anything, *e.g.*, the process with all failures, is seen as the smallest, least restrictive specification.

Remark 2.1.11. The notions defined above appear in different formulations in [137]. Their definition of stable failures refinement omits the clause for weak trace inclusion, and their definition of failures-divergences refinement replaces $\mathsf{failures}_\perp$ with $\mathsf{failures}$. This yields refinement relations different from the standard ones and neither relation seems to appear in the literature [55].

We conclude with a small example, illustrating the uses of, and differences between the various refinement relations.

Example 2.1.12. Consider the LTSs s_0 and u_0 of Example 2.1.8 again and the LTS t_0 depicted in between. We now consider s_0 to be the specification of a simplified automated teller machine.



In the specification s_0 the user can first request, by action REQ, an amount of twenty from the machine. The machine can then satisfy this request by either choosing to give twenty directly or by presenting two times ten to the user, which might vary depending on availability within the machine. Note that the distinction between user-initiated and response actions is only for the sake of the explanation and is not formally present in the LTS.

An implementation of this specification is *valid* if and only if it refines the specification in the required refinement semantics. Let us consider t_0 as a first implementation of this machine. The weak traces of t_0, consisting of the set $\{\varepsilon, \text{REQ}, \text{REQ}\,20\}$, are included in the specification and therefore $s_0 \sqsubseteq_{\mathsf{tr}} t_0$. Trace refinement is suitable for safety properties; for example the absence of infinite sequences of 10 or 20 actions without matching requests can be specified in such semantics. However, trace refinement does not preserve liveness properties. In particular, deadlocks are not preserved by trace refinement. In contrast, stable failures refinement *does* preserve deadlock freedom. For instance, the observation of the failure $(\text{REQ}\,20, \{\text{REQ}, 20, 10\})$ of t_0 leads to the *deadlocked* state t_2, *i.e.*, a state with no outgoing transitions. This failure is not among the failures that can be observed of state s_0. Consequently, $s_0 \not\sqsubseteq_{\mathsf{sfr}} t_0$.

The second implementation that we consider is u_0. The self-loop above state u_1 might indicate that the machine uses (repeated) polling via a potentially unstable connection to determine whether the user's bank account permits the requested withdrawal. Note that u_1 is not a stable state; all failures are thus of the shape $(\text{REQ}\,20\,\text{REQ}\,20\ldots, \{10, 20\})$ and these are permitted observations for the given specification s_0. Therefore, this implementation is a valid stable failures refinement of the given specification, *i.e.*, $s_0 \sqsubseteq_{\mathsf{sfr}} u_0$. The divergences $\text{REQ}\,\rho$ for sequences $\rho \in Act^*$ cause this implementation not to be valid under failures-divergences refinement, *i.e.*, $s_0 \not\sqsubseteq_{\mathsf{fdr}} u_0$. From the perspective of the user, it is indeed questionable whether u_0 constitutes a proper implementation, as she may perceive a divergence of the system as a deadlock. □

Note that stable failures refinement is a stronger relation than trace refinement; *i.e.*, whenever $\sqsubseteq_{\mathsf{sfr}}$ holds then also necessarily $\sqsubseteq_{\mathsf{tr}}$ holds. This does not hold the other way around as already shown in Example 2.1.12 where $s_0 \sqsubseteq_{\mathsf{tr}} t_0$ and $s_0 \not\sqsubseteq_{\mathsf{sfr}} t_0$. Furthermore, $\sqsubseteq_{\mathsf{fdr}}$ is incomparable to $\sqsubseteq_{\mathsf{tr}}$. In the preceding example, we have $u_0 \sqsubseteq_{\mathsf{fdr}} s_0$, but $u_0 \not\sqsubseteq_{\mathsf{tr}} s_0$ because $\text{REQ}\,10 \in \mathsf{weaktraces}(s_0)$ and $\text{REQ}\,10 \notin \mathsf{weaktraces}(u_0)$. But we also have $s_0 \sqsubseteq_{\mathsf{tr}} t_0$ and $s_0 \not\sqsubseteq_{\mathsf{fdr}} t_0$, where the latter fails because $(\text{REQ}\,20, \{\text{REQ}, 20, 10\}) \in \mathsf{failures}_{\perp}(t_0)$, but $(\text{REQ}\,20, \{\text{REQ}, 20, 10\}) \notin \mathsf{failures}_{\perp}(s_0)$. Similarly, $\sqsubseteq_{\mathsf{fdr}}$ is incomparable to $\sqsubseteq_{\mathsf{sfr}}$. For instance, we have $s_0 \sqsubseteq_{\mathsf{sfr}} u_0$ and $s_0 \not\sqsubseteq_{\mathsf{fdr}} u_0$ in Example 2.1.12, but also $u_0 \sqsubseteq_{\mathsf{fdr}} s_0$ and $u_0 \not\sqsubseteq_{\mathsf{sfr}} s_0$, where for the latter we observe that $(\text{REQ}, \{10\}) \in \mathsf{failures}(s_0)$ but $(\text{REQ}, \{10\}) \notin \mathsf{failures}(u_0)$.

2.2 Refinement Checking

In general, the set of weak traces, failures and divergences of an LTS can be infinite. Therefore, checking inclusion of these sets directly is not always viable. In [121, 122], an algorithm to decide refinement between two labelled transition systems is sketched. As a preprocessing step to this algorithm, all diverging states in both LTSs are marked. The algorithm then relies on exploring the product of a *normal form* representation of the specification, *i.e.*, the LTS that is to be refined, and the implementation.

For each state in this product it checks whether it can locally decide non-refinement of the implementation state with the normal form state. A state for which non-refinement holds is referred to as a *witness*. Following [122, 137] and specifically the terminology of [121], we formalise the product between LTSs that is explored by the procedure.

Definition 2.2.1 (Product)**.** Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. The *product* of $\mathscr{L}_1$ and $\mathscr{L}_2$, denoted by $\mathscr{L}_1 \ltimes \mathscr{L}_2$, is an LTS $(S, \iota, \rightarrow)$ such that $S = S_1 \times S_2$ and $\iota = (\iota_1, \iota_2)$. The transition relation $\rightarrow$ is the smallest relation such that for all $s_1, t_1 \in S_1$, and $s_2, t_2 \in S_2$ and $a \in Act$:

- If $s_2 \xrightarrow{\tau}_2 t_2$ then $(s_1, s_2) \xrightarrow{\tau} (s_1, t_2)$.
- If $s_1 \xrightarrow{a}_1 t_1$ and $s_2 \xrightarrow{a}_2 t_2$ then $(s_1, s_2) \xrightarrow{a} (t_1, t_2)$.

The proposition below relates the behaviours of two LTSs to the behaviours of their product.

Proposition 2.2.2. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs and let $\mathscr{L}_1 \ltimes \mathscr{L}_2 = (S, \iota, \rightarrow)$. For all states $(s_1, t_1), (s_2, t_2) \in S$ and all sequences $\rho \in Act^*$ it holds that $(s_1, s_2) \stackrel{\rho}{\Longrightarrow} (t_1, t_2)$ if and only if $s_1 \stackrel{\rho}{\twoheadrightarrow}_1 t_1$ and $s_2 \stackrel{\rho}{\Longrightarrow}_2 t_2$.

Proof. First, we can show that the statement holds for the empty sequence by induction on the length of sequences in τ^*. Using this we can prove the statement for all sequences in Act^* by induction on their length using the previous result in the base case. □

The normal form LTS of a given LTS is obtained using a typical subset construction as is common when determinising a transition system. A difference between determinisation and normalisation is that the former yields a transition system that preserves and reflects the set of weak traces of the given LTS. This is not the case for the latter, which may add weak traces not present in the original LTS. We first introduce the normal form LTS of a given LTS that is adequate for reducing the trace refinement and stable failures decision problems to a reachability problem.

Definition 2.2.3 (Normal form)**.** Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS. The normal form of $\mathscr{L}$ is the LTS $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$, where $S' = \mathscr{P}(S)$, $\iota' = \{s \in S \mid \iota \stackrel{\varepsilon}{\Longrightarrow} s\}$ and $\rightarrow'$ is defined as $U \stackrel{a}{\rightarrow}' V$ if and only if $V = \{t \in S \mid \exists s \in U : s \stackrel{a}{\Longrightarrow} t\}$ for all sets of states $U, V \subseteq S$ and actions $a \in Act$.

Notice that $\emptyset$ is a state in a normal form LTS. Furthermore, the normal form LTS is *deterministic*, *concrete* and *universal*.

Since a normal form LTS is concrete, all of its states are stable. The states of the original LTS comprising a normal form state may not be stable, however. When we need to reason about the stability and refusals of the set of states U in the LTS $\mathscr{L}$ underlying a normal form LTS, rather than the state U of the normal form LTS, we therefore write $[\![U]\!]_{\mathscr{L}}$ whenever we wish to stress that we refer to the set of states in $\mathscr{L}$ that comprise U.

The three lemmas stated below relate the set of weak traces of an LTS $\mathscr{L}$ to the set of traces of $\mathsf{norm}(\mathscr{L})$.

Lemma 2.2.4. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and states $U \in S'$ such that $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$, it holds that $\iota \stackrel{\rho}{\Longrightarrow} s$ for all $s \in [\![U]\!]_{\mathscr{L}}$.

Proof. Can be found in Appendix A.1. □

Lemma 2.2.5. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and for all states $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$, there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$.

Proof. Can be found in Appendix A.2. □

The lemma below clarifies the role of the state $\emptyset$ in $\mathsf{norm}(\mathscr{L})$.

Lemma 2.2.6. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ it holds that $\rho \notin \mathsf{weaktraces}(\mathscr{L})$ if and only if $\iota' \stackrel{\rho}{\twoheadrightarrow}' \emptyset$.

Proof. Can be found in Appendix A.3. □

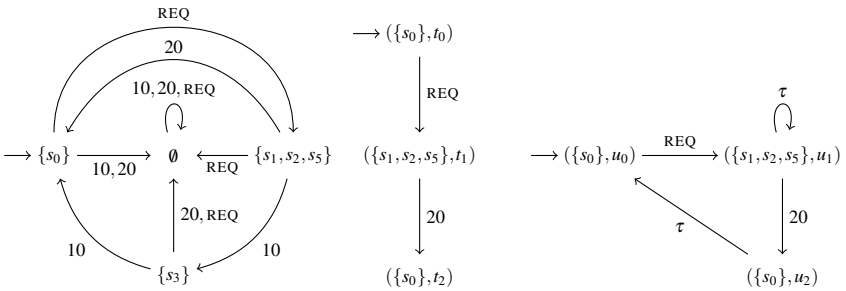
The structure explored by the refinement checking procedure of [121, 122] for two LTSs $\mathscr{L}_1$ and $\mathscr{L}_2$ is the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ in case of trace refinement and stable failures refinement. For these structures the related witnesses, where the reachability of such a witness indicates non-refinement, are then as follows:

Definition 2.2.7 (Witness)**.** Let $\mathscr{L}_1$ and $\mathscr{L}_2$ be LTSs. A state pair (U, s) of product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$:

- is called a *TR-witness* if and only if $U = \emptyset$.
- is called an *SF-witness* if and only if at least one of the following conditions hold:
  - $U = \emptyset$.
  - $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$.

We illustrate the notion of a witness, and in particular the relation between the reachability of a witness and the (violation of) the corresponding refinement relation by means of a small example.

Example 2.2.8. Consider the specification s_0 and the two implementations t_0 and u_0 as presented in Example 2.1.12 again. In the figure below the (reachable part of the) normal form LTS of s_0 is depicted on the left, the product $\mathsf{norm}(s_0) \ltimes t_0$ is shown in the middle and the product $\mathsf{norm}(s_0) \ltimes u_0$ is shown on the right.



We observe that both $\mathsf{norm}(s_0) \ltimes t_0$ and $\mathsf{norm}(s_0) \ltimes u_0$ contain no TR-witnesses. In Example 2.1.12 we had already established that both $s_0 \sqsubseteq_{\mathsf{tr}} t_0$ and $s_0 \sqsubseteq_{\mathsf{tr}} u_0$. For the product $\mathsf{norm}(s_0) \ltimes t_0$, the state $(\{s_0\}, t_2)$ is reachable and an SF-witness since $\mathsf{stable}(t_2)$ and $\{10, 20, \mathrm{REQ}\} \in \mathsf{refusals}(t_2)$ but $\{10, 20, \mathrm{REQ}\} \notin \mathsf{refusals}(\{s_0\})$. Intuitively, the product encodes that $\mathrm{REQ}\,20$ is a weak trace in both LTSs $\mathsf{norm}(s_0)$ and t_2 that reaches $\{s_0\}$ and t_2 respectively. Similarly, the normal form relates this weak trace to the reachability of s_0 from s_0. Therefore, this witness indicates that $(\mathrm{REQ}\,20, \{10, 20, \mathrm{REQ}\})$ is a failure of t_0, but not a failure of s_0, which establishes a violation of stable failures refinement. Finally, we can observe that $\mathsf{norm}(s_0) \ltimes u_0$ contains no SF-witnesses. □

The following lemmas formalise that trace refinement can be decided by checking reachability of a TR-witness in the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Note that this result, and the related result for stable failures refinement, was already established in the literature; for instance, the relation between an SF-witness and the corresponding refinement relation can be found in [121]. However, the definitions in that paper are

not explicit and the proof of this correspondence is only sketched. Therefore, we here provide detailed proofs of these results.

Lemma 2.2.9. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ holds then no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. Suppose that $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ holds. Therefore, it follows that $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. Now assume that there is a reachable TR-witness $(\emptyset, s)$ in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. We show that this leads to a contradiction. As the pair $(\emptyset, s)$ is reachable there is a weak trace $\rho \in Act^*$ such that $\iota \stackrel{\rho}{\Longrightarrow} (\emptyset, s)$ where ι is the initial state of $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. From Proposition 2.2.2 it follows that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and $\emptyset$ is reachable by following ρ in $\mathsf{norm}(\mathscr{L}_1)$. Therefore, $\rho \in \mathsf{weaktraces}(\mathscr{L}_2)$ and from Lemma 2.2.6 it follows that $\rho \notin \mathsf{weaktraces}(\mathscr{L}_1)$. This contradicts our assumption that $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. Hence, no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. □

Lemma 2.2.10. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ then $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$.

Proof. Suppose that no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Again, we prove this by contradiction. Assume that $\mathscr{L}_1 \not\sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ holds. This means that $\mathsf{weaktraces}(\mathscr{L}_2) \not\subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. Pick a weak trace $\rho \in \mathsf{weaktraces}(\mathscr{L}_2)$ such that $\rho \notin \mathsf{weaktraces}(\mathscr{L}_1)$. Then there is a state $s \in S_2$ for which $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$. By Lemma 2.2.6 it holds that ρ leads to the empty set in $\mathsf{norm}(\mathscr{L}_1)$. By Proposition 2.2.2 the pair $(\emptyset, s)$ is then a reachable TR-witness in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Contradiction. □

Theorem 2.2.11. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. Then $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ holds if and only if no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. This follows directly from Lemmas 2.2.9 and 2.2.10. □

Next, we formalise the relation between stable failures refinement and the reachability of an SF-witness. In the proofs for the next two lemmas, we exploit Theorem 2.2.11 and the fact that stable failures refinement is stronger than trace refinement.

Lemma 2.2.12. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$ holds then no SF-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. Suppose that $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$ holds. Therefore, both $\mathsf{failures}(\mathscr{L}_2) \subseteq \mathsf{failures}(\mathscr{L}_1)$ and $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. Now assume that there is a reachable SF-witness (U, s) in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. We show that this leads to a contradiction. For (U, s) to be an SF-witness it holds that $U = \emptyset$ or both $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$. However, since $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$ it follows that

$\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ and, hence, by Theorem 2.2.11, no TR-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Consequently, $U \neq \emptyset$, and therefore it must be that $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$ hold.

As the pair (U,s) is reachable there is a weak trace $\rho \in Act^*$ such that $\iota \stackrel{\rho}{\Longrightarrow} (U,s)$ where ι is the initial state of $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. From Proposition 2.2.2 it follows that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and U is reachable by following ρ in $\mathsf{norm}(\mathscr{L}_1)$. Since $\mathsf{stable}(s)$ and $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$, it follows that there must be a failure $(\rho,X) \in \mathsf{failures}(\mathscr{L}_2)$ where s can stably refuse $X \in \mathsf{refusals}(s)$, but $X \notin \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$. Let (ρ,X) be such. By Lemmas 2.1.4 and 2.2.5 it follows for all states $t \in S_1$ where $\iota_1 \stackrel{\rho}{\Longrightarrow}_1 t$ that $t \in \llbracket U \rrbracket_{\mathscr{L}_1}$. For each stable t it holds that $X \notin \mathsf{refusals}(t)$, because $X \notin \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$. Therefore, we conclude that $(\rho,X) \notin \mathsf{failures}(\mathscr{L}_1)$, which contradicts $\mathsf{failures}(\mathscr{L}_2) \subseteq \mathsf{failures}(\mathscr{L}_1)$. We conclude that the state (U,s) cannot be an SF-witness. □

Lemma 2.2.13. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If no SF-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ then $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$ holds.

Proof. Suppose that no SF-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Towards a contradiction, assume that $\mathscr{L}_1 \not\sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$. By definition of the stable failures refinement this means that $\mathsf{failures}(\mathscr{L}_2) \not\subseteq \mathsf{failures}(\mathscr{L}_1)$ or $\mathsf{weaktraces}(\mathscr{L}_2) \not\subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. If $\mathsf{weaktraces}(\mathscr{L}_2) \not\subseteq \mathsf{weaktraces}(\mathscr{L}_1)$ then $\mathscr{L}_1 \not\sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$ and so, by Theorem 2.2.11, there must be a reachable TR-witness $(\emptyset,s)$, and, therefore, also a reachable SF-witness. Contradiction.

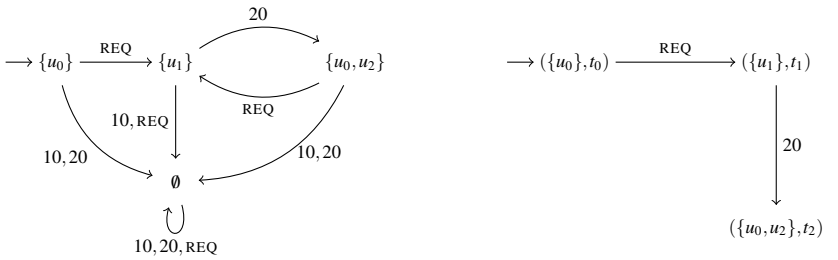
Therefore, $\mathsf{failures}(\mathscr{L}_2) \not\subseteq \mathsf{failures}(\mathscr{L}_1)$ and $\mathsf{weaktraces}(\mathscr{L}_2) \subseteq \mathsf{weaktraces}(\mathscr{L}_1)$. Pick a failure $(\rho,X) \in \mathsf{failures}(\mathscr{L}_2)$ such that $(\rho,X) \notin \mathsf{failures}(\mathscr{L}_1)$. Since it holds that $(\rho,X) \in \mathsf{failures}(\mathscr{L}_2)$, there is a stable state $s \in S_2$ such that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and $X \in \mathsf{refusals}(s)$. Since $(\rho,X) \in \mathsf{failures}(\mathscr{L}_2)$, also $\rho \in \mathsf{weaktraces}(\mathscr{L}_2)$, and therefore also $\rho \in \mathsf{weaktraces}(\mathscr{L}_1)$. By Lemmas 2.1.4 and 2.2.5 weak trace ρ leads to a unique state U in $\mathsf{norm}(\mathscr{L}_1)$ such that for all states $t \in S_1$ with $\iota_1 \stackrel{\rho}{\Longrightarrow}_1 t$ it holds that $t \in \llbracket U \rrbracket_{\mathscr{L}_1}$. For each stable t it holds that $X \notin \mathsf{refusals}(t)$, because $(\rho,X) \notin \mathsf{failures}(\mathscr{L}_1)$. Therefore, by Lemma 2.2.4 it follows that $X \notin \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$. Hence, $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\llbracket U \rrbracket_{\mathscr{L}_1})$. By Proposition 2.2.2 the pair (U,s) is reachable and it is an SF-witness by definition. Contradiction. □

Theorem 2.2.14. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. Then $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$ holds if and only if no SF-witness is reachable in $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. This follows directly from Lemmas 2.2.12 and 2.2.13. □

One may be inclined to believe that the normal form LTS can also be used to reduce the failures-divergences refinement decision problem to a reachability problem. This is, however, not the case as the following example illustrates.

Example 2.2.15. Reconsider the LTSs t_0 and u_0 of Example 2.1.12. Note that t_0 is a correct failures-divergences refinement of u_0, *i.e.*, $u_0 \sqsubseteq_{\mathsf{fdr}} t_0$. The divergences $\mathrm{REQ}\,\rho$, for $\rho \in Act^*$, of u_0 result in specification u_0 permitting all these sequences.



Consider the normal form $\mathsf{norm}(u_0)$ shown above on the left. The pair $(\{u_0, u_2\}, t_2)$ in the product $\mathsf{norm}(u_0) \ltimes t_0$ shown on the right, is thus problematic for the analysis of failures-divergence refinement, as the reachability of this pair might incorrectly indicate a violation of $u_0 \sqsubseteq_{\mathsf{fdr}} t_0$. In turn, this suggests that in the reachability analysis of the product, states beyond those reached via a trace constituting a divergence should not be considered candidate witnesses. □

Our solution is to modify the construction of the normal form LTS for failures-divergences refinement as follows.

Definition 2.2.16 (Failures-divergences normal form)**.** Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS. The failures-divergences normal form of $\mathscr{L}$ is the LTS $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$, where $S' = \mathscr{P}(S)$, $\iota' = \{s \in S \mid \iota \stackrel{\varepsilon}{\Longrightarrow} s\}$ and $\rightarrow'$ is defined as $U \stackrel{a}{\rightarrow}' V$ if and only if $\neg(\exists s \in U : s\Uparrow)$ and $V = \{t \in S \mid \exists s \in U : s \stackrel{a}{\Longrightarrow} t\}$ for all sets of states $U, V \subseteq S$ and actions $a \in Act$.

Notice that $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L})$ yields a subgraph of $\mathsf{norm}(\mathscr{L})$. As a result, several properties that we established for norm carry over to $\mathsf{norm}_{\mathsf{fdr}}$. For instance, $\mathsf{norm}_{\mathsf{fdr}}$ yields LTSs that are deterministic and concrete. However, contrary to LTSs obtained via norm, LTSs obtained via $\mathsf{norm}_{\mathsf{fdr}}$ are not guaranteed to be universal. In particular, a weak trace $\rho \in \mathsf{weaktraces}(\mathscr{L})$ is not guaranteed to be preserved in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L})$ if it is a divergence. Consequently, Lemma 2.2.6, which is essential for Theorems 2.2.11 and 2.2.14, no longer holds in its full generality. We show, however, that for failures-divergence refinement a slightly different relation between an LTS and its normal form is sufficient for establishing a theorem that is similar in spirit to the aforementioned theorems.

Lemma 2.2.17. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and states $U \in S'$ such that $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$ it holds that $\iota \stackrel{\rho}{\Longrightarrow} s$ for all $s \in [\![U]\!]_{\mathscr{L}}$.

Proof. Along the same lines as the proof of Lemma 2.2.4. □

We mentioned that divergences are not necessarily preserved (as traces) by the normalisation. In fact, we can be more specific: only *minimal* divergences are preserved in the normal form LTS. The *minimal* divergences of a state $s \in S$, denoted by $\mathsf{divergences}_{\mathsf{min}}(s)$, is the largest subset of $\mathsf{divergences}(s)$ containing all divergencesa $\rho \in \mathsf{divergences}(s)$ for which there is no strict prefix of ρ in $\mathsf{divergences}(s)$. For an LTS $\mathscr{L} = (S, \iota, \rightarrow)$ we define $\mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$ to be $\mathsf{divergences}_{\mathsf{min}}(\iota)$.

Lemma 2.2.18. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ such that either $\rho \notin \mathsf{divergences}(\mathscr{L})$ or $\rho \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$ and for all states $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$ there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$.

Proof. Can be found in Appendix A.4. □

Lemma 2.2.19. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and states $U \in S'$ it holds that if $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$ and not $[\![U]\!]_{\mathscr{L}}\Uparrow$ then $\rho \notin \mathsf{divergences}(\mathscr{L})$.

Proof. Can be found in Appendix A.5. □

Lemma 2.2.20. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ it holds that $\rho \notin (\mathsf{divergences}(\mathscr{L}) \cup \mathsf{weaktraces}(\mathscr{L}))$ if and only if $\iota' \stackrel{\rho}{\twoheadrightarrow}' \emptyset$.

Proof. Can be found in Appendix A.6. □

For failures-divergences refinement the state space of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ is explored for a witness, where reachability of such a witness also indicates non-refinement. This witness is defined as follows:

Definition 2.2.21 (Failures-divergences witness)**.** Let $\mathscr{L}_1$ and $\mathscr{L}_2$ be two LTSs. A state (U, s) of the product $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ is called an *FD-witness* if and only if $[\![U]\!]_{\mathscr{L}_1}\Uparrow$ does not hold and at least one of the following conditions hold:

- $U = \emptyset$.
- $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$.
- $s\Uparrow$.

We next formalise the correspondence between failures-divergences refinement and the reachability of an FDR-witness. In the proof of the following theorem we cannot easily use Theorem 2.2.11 because $\sqsubseteq_{\mathsf{fdr}}$ is incomparable with both $\sqsubseteq_{\mathsf{tr}}$ and $\sqsubseteq_{\mathsf{sfr}}$.

Lemma 2.2.22. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$ holds then no FD-witness is reachable in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. Assume that $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$. We then have that $\mathsf{failures}_\perp(\mathscr{L}_2) \subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$ and $\mathsf{divergences}(\mathscr{L}_2) \subseteq \mathsf{divergences}(\mathscr{L}_1)$. Towards a contradiction, assume there is an FD-witness (U,s) in $\mathsf{reachable}(\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2)$. Let ι be the initial state of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$, and let $\rho \in \mathsf{weaktraces}(\iota)$ be such that $\iota \stackrel{\rho}{\Longrightarrow} (U,s)$. From the assumption that (U,s) is an FD-witness it follows that not $[\![U]\!]_{\mathscr{L}_1}\Uparrow$ and from Lemma 2.2.19 it follows that $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$. By Proposition 2.2.2 it holds that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and U is reachable by following ρ in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1)$. Moreover, for (U,s) to be an FD-witness, at least one of the following must also hold: $U = \emptyset$, $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$, or $s\Uparrow$. We therefore distinguish these three cases:

- Case $U = \emptyset$. We can assume that $s\Uparrow$ does not hold, as this is handled by another case. Then it follows that there is a state $t \in S_2$ such that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s \stackrel{\varepsilon}{\Longrightarrow}_2 t$ and $\mathsf{stable}(t)$. Let t be such. Consequently, $(\rho, X) \in \mathsf{failures}_\perp(\mathscr{L}_2)$ for some $X \in \mathsf{refusals}(t)$. By Lemma 2.2.20 it holds that the weak trace ρ reaching the empty set in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1)$ is not a weak trace of $\mathscr{L}_1$. Together with $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$ it follows, for all possible refusal sets $Y \subseteq Act$, that $(\rho, Y) \notin \mathsf{failures}_\perp(\mathscr{L}_1)$, and so, in particular, $(\rho, X) \in \mathsf{failures}_\perp(\mathscr{L}_1)$ which contradicts our assumption that $\mathsf{failures}_\perp(\mathscr{L}_2) \subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$.
- Case $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$. From $\mathsf{stable}(s)$ and the reachability of state s it follows that $\mathsf{failures}_\perp(\mathscr{L}_2)$ is not empty. Pick a failure $(\rho, X) \in \mathsf{failures}_\perp(\mathscr{L}_2)$ where s can stably refuse $X \in \mathsf{refusals}(s)$, but $X \notin \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$. By Lemmas 2.1.4 and 2.2.18 it follows for all states $t \in S_1$ where $\iota_1 \stackrel{\rho}{\Longrightarrow}_1 t$ that $t \in [\![U]\!]_{\mathscr{L}_1}$. For each stable t it holds that $X \notin \mathsf{refusals}(t)$, because $X \notin \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$. Due to the previous case, we may assume that $U \neq \emptyset$. Then from $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$ and $U \neq \emptyset$ it follows that $(\rho, X) \notin \mathsf{failures}_\perp(\mathscr{L}_1)$, which leads to a contradiction with the assumption that $\mathsf{failures}_\perp(\mathscr{L}_2) \subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$.
- Case $s\Uparrow$. Since $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and $s\Uparrow$, also $\rho \in \mathsf{divergences}(\mathscr{L}_2)$. However, by $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$ this contradicts the assumption that $\mathsf{divergences}(\mathscr{L}_2) \subseteq \mathsf{divergences}(\mathscr{L}_1)$. □

Lemma 2.2.23. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. If no FD-witness is reachable in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ then $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$ holds.

Proof. Assume that no FD-witness is reachable in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Again, we prove this by contradiction. Assume that $\mathscr{L}_1 \not\sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$. By definition of failures-divergences refinement this means that $\mathsf{failures}_\perp(\mathscr{L}_2) \not\subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$ or it might be that $\mathsf{divergences}(\mathscr{L}_2) \not\subseteq \mathsf{divergences}(\mathscr{L}_1)$. Hence, there are two cases to consider:

- Case $\mathsf{divergences}(\mathscr{L}_2) \not\subseteq \mathsf{divergences}(\mathscr{L}_1)$. Pick a diverging weak trace $\rho \in \mathsf{divergences}(\mathscr{L}_2)$ such that $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$. In this case there is a prefix of ρ, which we call σ, that leads to a diverging state $s \in S_2$ such that $\iota_2 \stackrel{\sigma}{\Longrightarrow}_2 s$. However, by the assumption that $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$ we know that all states $t \in S_1$ reached by following σ are not diverging. By Lemma 2.2.17 we know that all $t \in U$ can be reached by following σ. Therefore state pair (U,s) is an FD-witness, because $s\Uparrow$ but not $[\![U]\!]_{\mathscr{L}_1}\Uparrow$. Contradiction.
- Case $\mathsf{failures}_\perp(\mathscr{L}_2) \not\subseteq \mathsf{failures}_\perp(\mathscr{L}_1)$. By the previous case, we may, moreover, assume that $\mathsf{divergences}(\mathscr{L}_2) \subseteq \mathsf{divergences}(\mathscr{L}_1)$. Pick any failure $(\rho,X) \in \mathsf{failures}_\perp(\mathscr{L}_2)$ such that $(\rho,X) \notin \mathsf{failures}_\perp(\mathscr{L}_1)$. Observe that weak trace $\rho \notin \mathsf{divergences}(\mathscr{L}_1)$ (and as such $\rho \notin \mathsf{divergences}(\mathscr{L}_2)$) as otherwise no such (ρ,X) exists by definition of $\mathsf{failures}_\perp(\mathscr{L}_1)$. Since $(\rho,X) \in \mathsf{failures}_\perp(\mathscr{L}_2)$, there is a stable state $s \in S_2$ such that $\iota_2 \stackrel{\rho}{\Longrightarrow}_2 s$ and $X \in \mathsf{refusals}(s)$. We distinguish whether weak trace ρ is among the weak traces of $\mathscr{L}_1$ or not:
  - Case $\rho \notin \mathsf{weaktraces}(\mathscr{L}_1)$. By Lemma 2.2.20 this means that ρ is a trace leading to the empty set in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1)$. By Proposition 2.2.2, a pair $(\emptyset,s)$ is then reachable in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. But then that pair is a reachable FD-witness. Contradiction.
  - Case $\rho \in \mathsf{weaktraces}(\mathscr{L}_1)$. Recall that $(\rho,X) \notin \mathsf{failures}_\perp(\mathscr{L}_1)$ by assumption. By Lemmas 2.1.4 and 2.2.18 there is a unique state V of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1)$ reachable via weak trace ρ such that for all $t \in S_1$ where $\iota_1 \stackrel{\rho}{\Longrightarrow}_1 t$, it holds that $t \in [\![V]\!]_{\mathscr{L}_1}$. For each stable state t it holds that $X \notin \mathsf{refusals}(t)$, because $(\rho,X) \in \mathsf{failures}_\perp(\mathscr{L}_1)$. Therefore, by Lemma 2.2.17 it follows that $X \notin \mathsf{refusals}([\![V]\!]_{\mathscr{L}_1})$. Therefore $\mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![V]\!]_{\mathscr{L}_1})$. By Proposition 2.2.2 the pair (V,s) is reachable and it is an FD-witness by definition. Contradiction. □

Theorem 2.2.24. Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs. Then $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$ holds if and only if no FD-witness is reachable in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof. This follows directly from Lemmas 2.2.22 and 2.2.23. □

Example 2.2.25. Consider the LTSs t_0 and u_0 of Example 2.1.12 once more. As we noted in Example 2.2.15, t_0 is a correct failures-divergences refinement of u_0. In the figure below the normal form $\mathsf{norm}_{\mathsf{fdr}}(u_0)$ is shown on the left and the product $\mathsf{norm}_{\mathsf{fdr}}(u_0) \ltimes t_0$ is shown on the right.

Note that the pair $(\{u_0,u_2\},t_2)$, which was reachable in the product $\mathsf{norm}(u_0) \ltimes t_0$, is no longer reachable in the product $\mathsf{norm}_{\mathsf{fdr}}(u_0) \ltimes t_0$. In fact, no FD-witness is reachable in the latter product, thus confirming that indeed $u_0 \sqsubseteq_{\mathsf{fdr}} t_0$. □

2.3 Antichain Algorithms for Refinement Checking

Notice that Theorems 2.2.11, 2.2.14 and 2.2.24 provide the basis for straightforward algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement: one can explore the product of the normalised specification and the impementation, looking for a witness *on-the-fly*. In these algorithms, the normalisation of the specification LTS dominates the theoretical worst-case run time complexity of the algorithms. While refinement checking itself is a PSPACE-hard problem, in practice, the problem can often be solved quite effectively. Nevertheless, as observed in [137], antichains provide room for improvement by potentially reducing the number of states of the normal form LTS of the specification that must be checked.

An antichain is a set $\mathscr{A} \subseteq X$ of a partially ordered set $(X,\leq)$ in which all distinct $x,y \in \mathscr{A}$ are incomparable: neither $x \leq y$ nor $y \leq x$. Given a partially ordered set $(X,\leq)$ and an antichain $\mathscr{A}$, the membership test, denoted by $\Subset$, checks whether $\mathscr{A}$ 'contains' an element x; that is, $x \Subset \mathscr{A}$ holds true if and only if there is some $y \in \mathscr{A}$ such that $y \leq x$. We write $Y \Subset^{\forall} \mathscr{A}$ iff $y \Subset \mathscr{A}$ for all $y \in Y$. Antichain $\mathscr{A}$ can be extended by inserting an element $x \in X$, denoted $\mathscr{A} \Cup x$, which is defined as the set $\{y \mid y = x \vee (y \in \mathscr{A} \wedge x \not\leq y)\}$. Note that this operation only yields an antichain whenever $x \notin \mathscr{A}$.

As [137, 1] suggest, the state space of the product $(S,\iota,\rightarrow)$ between a normal form of LTS $\mathscr{L}_1$ and the LTS $\mathscr{L}_2$ induces a partially ordered set as follows. For $(U,s),(V,t) \in S$, define $(U,s) \leq (V,t)$ iff $s = t$ and $[\![U]\!]_{\mathscr{L}_1} \subseteq [\![V]\!]_{\mathscr{L}_1}$. Then the set $(S,\leq)$ is a partially ordered set. The fundamental property underlying the reason why an antichain approach to refinement checking works is expressed by the following claim (which we repeat as Proposition 2.4.7, and prove in Section 2.4), stating that the traces of any state (V,s) in the product can be executed from all states smaller than (V,s). Notice that this property relies on the fact that the empty set is included as a

state in the normal form LTS.

Claim 2.3.1. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ and for every sequence $\sigma \in Act_\tau{}^*$ such that $(V,s) \stackrel{\sigma}{\twoheadrightarrow} (V',t)$ there is a state (U',t) such that $(U,s) \stackrel{\sigma}{\twoheadrightarrow} (U',t)$ and $(U',t) \leq (V',t)$.

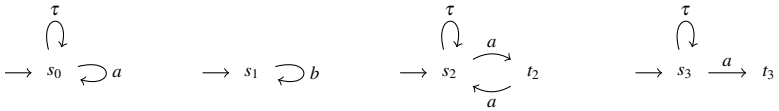
The main idea of the antichain algorithms is now as follows: the set of states of the product that have been explored are recorded in an antichain rather than a set. Whenever a new state of the product is found that is already included in the antichain (w.r.t. the membership test $\Subset$), further exploration of that state is unnecessary, thereby pruning the state space of the product. While the proposition stated above suggests this is sound for trace refinement, it is not immediate that doing so is also sound for refusals and divergences.

Based on the above informal reasoning, [137] presents antichain algorithms that intend to check for trace refinement, stable failures refinement and failures-divergences refinement. Before we discuss these algorithms in more detail, see Algorithms 1-3, we here present their pseudocode for the sake of completeness; in the remainder of this chapter, we refer to these as the *original* algorithms.

Remark 2.3.2. For the implementation of refusals in Algorithms 2 and 3 we followed the definition of $\mathsf{refusals}$ provided in [137]. This definition differs subtly from Definition 2.1.5, by defining, for any (not necessarily stable) state s, $\mathsf{refusals}(s) = \{X \mid \exists s' \in S : (s \stackrel{\varepsilon}{\Longrightarrow} s' \wedge \mathsf{stable}(s') \wedge X \subseteq Act \setminus \mathsf{enabled}(s'))\}$ and $\mathsf{refusals}(U) = \{X \mid \exists s \in U : X \in \mathsf{refusals}(s)\}$ for $U \subseteq S$.

Let us stress that Algorithm 1 correctly decides trace refinement and Algorithm 2 correctly decides stable failures refinement. However, Algorithm 3 fails to correctly decide failures-divergences refinement. Moreover, it is interesting to note that Algorithms 2 and 3 fail to decide the *non-standard* relations used in [137], see also the discussion in Remark 2.1.11. All three issues are illustrated by the example below.

Example 2.3.3. Consider the four transition systems depicted below.



Let us first observe that Algorithm 2 correctly decides that $s_1 \sqsubseteq_{\mathsf{sfr}} s_0$ does not hold, which follows from a violation of $\mathsf{weaktraces}(s_0) \subseteq \mathsf{weaktraces}(s_1)$. Next, observe that we have $s_0 \sqsubseteq_{\mathsf{fdr}} s_1$, since the divergence of the root state s_0 implies chaotic behaviour of s_0 and, hence, any system refines such a system. It is not hard to see, however, that Algorithm 3 returns *false*, wrongly concluding that $s_0 \not\sqsubseteq_{\mathsf{fdr}} s_1$.

Algorithm 1 Antichain-based trace refinement algorithm presented in [137]. The algorithm returns *true* if and only if $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in trace semantics.

```
1:  procedure REFINES-TRACE(𝓛1, 𝓛2)
2:      let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
3:      let antichain ← ∅
4:      while working ≠ ∅ do
5:          pop (spec, impl) from working
6:          antichain ← antichain ⩔ (spec, impl)
7:          for impl -a->_2 impl' do
8:              if a = τ then
9:                  spec' ← spec
10:             else
11:                 spec' ← {s' ∈ S1 | ∃s ∈ spec : s ⟹a_1 s'}
12:             if spec' = ∅ then
13:                 return false
14:             if (spec', impl') ∉ antichain then
15:                 push (spec', impl') into working
16:     return true
```

With respect to the non-standard refinement relations defined in [137], see also Remark 2.1.11, we observe the following. Since s_0 is not stable, we have $\mathsf{failures}(s_0) = \emptyset$ and hence $\mathsf{failures}(s_0) \subseteq \mathsf{failures}(s_1)$. Consequently, stable failures refinement as defined in [137] should hold, but as we already concluded above, the algorithm returns *false* when checking for $s_1 \sqsubseteq_{\mathsf{sfr}} s_0$. Next, observe that the algorithm returns *true* when checking for $s_2 \sqsubseteq_{\mathsf{fdr}} s_3$. The reason is that for the pair $(\{s_2\}, s_3)$, it detects that state s_3 diverges and concludes that since also the normal form state of the specification $\{s_2\}$ diverges, it can terminate the iteration and return *true*. This is a consequence of splitting the divergence tests over two **if**-statements in lines 7 and 8. According to the failures-divergences refinement of [137], however, the algorithm should return *false*, since $\mathsf{failures}(s_3) \subseteq \mathsf{failures}(s_2)$ fails to hold: we have $(a, \{a\}) \in \mathsf{failures}(s_3)$ but not $(a, \{a\}) \in \mathsf{failures}(s_2)$. □

Notice that each algorithm explores the product between the normal form of a specification, and an implementation in a depth-first, on-the-fly manner. While depth-first search is typically used for detecting divergences, [121] states a number of reasons for running a refinement check in a breadth-first manner. Indeed, a compelling argument in favour of using a breadth-first search is conciseness of the counterexample in case of a non-refinement.

Each algorithm can be made to run in a breadth-first fashion simply by using a FIFO *queue* rather than a stack as the data structure for *working*. However, our implementations of these algorithms suffer from severely degraded performance. The

Algorithm 2 Antichain-based stable failures refinement algorithm presented in [137]. The algorithm returns *true* if and only if $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in stable failures semantics.

```
 1: procedure REFINES-STABLE-FAILURES(𝓛1, 𝓛2)
 2:     let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
 3:     let antichain ← ∅
 4:     while working ≠ ∅ do
 5:         pop (spec, impl) from working
 6:         antichain ← antichain ⩒ (spec, impl)
 7:         if refusals(impl) ⊈ refusals(spec) then
 8:             return false
 9:         for impl −a→_2 impl′ do
10:             if a = τ then
11:                 spec′ ← spec
12:             else
13:                 spec′ ← {s′ ∈ S1 | ∃s ∈ spec : s ⟹a_1 s′}
14:             if spec′ = ∅ then
15:                 return false
16:             if (spec′, impl′) ∉ antichain then
17:                 push (spec′, impl′) into working
18:     return true
```

performance degradation can be traced back to the following three additional problems in the original algorithms, which also are present (albeit less pronounced in practice) when utilising a depth-first exploration:

1. The refusal check on line 7 of Algorithm 2 (and line 11 of Algorithm 3) is also performed for unstable states, which, combined with the definition of refusals in [137] (see also Remark 2.3.2), results in a repeated, potentially expensive, search for stable states;

2. In all three algorithms, duplicate pairs might be added to *working* since *working* is filled with all successors of $(spec, impl)$ that fail the *antichain* membership test, regardless of whether these pairs are already scheduled for exploration, *i.e.*, included in *working*, or not;

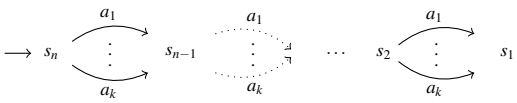
3. In all three algorithms, contrary to the explicit claim in [137, Section 2.2] the variable *antichain* is not guaranteed to be an antichain.

The first problem is readily seen to lead to undesirable overhead. The second and third problem are more subtle. We first illustrate the second problem on Algorithm 1: the following example shows a case where the algorithm stores an excessive number of pairs in *working*. Note that the two other algorithms suffer from the same phenomenon.

Algorithm 3 Antichain-based failures-divergences refinement algorithm presented in [137]. The algorithm is claimed to return *true* iff $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in failures-divergences semantics.

```
 1: procedure REFINES-FAILURES-DIVERGENCES(𝓛1, 𝓛2)
 2:     let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
 3:     let antichain ← ∅
 4:     while working ≠ ∅ do
 5:         pop (spec, impl) from working
 6:         antichain ← antichain ⩏ (spec, impl)
 7:         if impl⇑ then
 8:             if not spec⇑ then
 9:                 return false
10:         else
11:             if refusals(impl) ⊈ refusals(spec) then
12:                 return false
13:             for impl -a->_2 impl' do
14:                 if a = τ then
15:                     spec' ← spec
16:                 else
17:                     spec' ← {s' ∈ S1 | ∃s ∈ spec : s ⟹a_1 s'}
18:                 if spec' = ∅ then
19:                     return false
20:                 if (spec', impl') ∉ antichain then
21:                     push (spec', impl') into working
22:     return true
```

Example 2.3.4. Consider the family of LTSs $\mathscr{L}_n^k = (S_n, \iota_n, \rightarrow_n)$ with states $S_n = \{s_1, \ldots, s_n\}$, transitions $s_i \xrightarrow{a_j}_n s_{i-1}$ for all $1 \leq j \leq k$, $1 < i \leq n$ and $\iota_n = s_n$; see also the transition system depicted below. Note that each LTS that belongs to this family is completely deterministic and concrete.



Each labelled transition system in this class has n states and $k \cdot (n-1)$ transitions. Suppose one checks for trace refinement between an implementation and specification both of which are given by $\mathscr{L}_n^k$; *i.e.*, we test for $\mathscr{L}_n^k \sqsubseteq_{\mathsf{tr}} \mathscr{L}_n^k$.

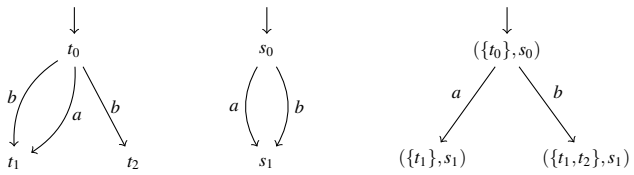
Using a depth-first search, Algorithm 1 will add the state reachable via a single step once for every action, because $(\{s_{i+1}\}, s_{i+1})$ is only added to *antichain* after $(\{s_i\}, s_i)$ has finished exploring its outgoing transitions. This occurs in every state, because the state reached via such a transition was not visited before. Hence, *working* contains exactly $i \cdot (k-1) + 1$ pairs at the end of the i-th iteration, resulting in a maximum

working stack size of $\mathcal{O}(n \cdot k)$ entries. At the end of the n-th iteration *antichain* contains all reachable pairs of the product, *i.e.*, *antichain* is equal to $(\{s_i\}, s_i)$ for all $1 \leq i \leq n$. Emptying *working* after the n-th iteration involves k antichain membership tests per entry. Consequently, $\mathcal{O}(n \cdot k^2)$ antichain membership tests are required to check $\mathscr{L}_n^k \sqsubseteq_{\mathsf{tr}} \mathscr{L}_n^k$.

The breadth-first variant of Algorithm 1 also adds the state reachable via a single step once for every action for the same reason as the depth-first variant. However, now $(\{s_{i+1}\}, s_{i+1})$ is only added to the *antichain* after *all* k copies of $(\{s_i\}, s_i)$ are taken from the FIFO queue *working*. Therefore each entry in *working* adds k elements before it is added to *antichain*, resulting in a maximum queue size of $\mathcal{O}(k^n)$ at state $(\{s_1\}, s_1)$. Emptying *working* results in $\mathcal{O}(k^{n+1})$ antichain membership tests. □

Finally, the example below illustrates the third problem of the algorithms, *viz.*, the violation of the antichain property. We again illustrate the problem on the most basic of all three algorithms, *viz.*, Algorithm 1. Note that this violation does not influence the result of antichain membership tests, but it can have an effect on the size of the antichain which in turn leads to overhead.

Example 2.3.5. Consider the two left-most labelled transition systems depicted below, along with the (normal form) product (the LTS on the right).



Algorithm 1 starts with *working* containing pair $(\{t_0\}, s_0)$ and $\mathit{antichain} = \emptyset$. Inside the loop, the pair $(\{t_0\}, s_0)$ is popped from *working* and added to *antichain*. The successors of the pair $(\{t_0\}, s_0)$ are the pairs $(\{t_1\}, s_1)$ and $(\{t_1, t_2\}, s_1)$. Since *antichain* contains neither of these, both successors are added to *working* in line 15. Next, popping $(\{t_1\}, s_1)$ from *working* and adding this pair to *antichain* results in *antichain* consisting of the set $\{(\{t_0\}, s_0), (\{t_1\}, s_1)\}$. In the final iteration of the algorithm, the pair $(\{t_1, t_2\}, s_1)$ is popped from *working* and added to *antichain*, resulting in the set $\{(\{t_0\}, s_0), (\{t_1\}, s_1), (\{t_1, t_2\}, s_1)\}$. Clearly, since $(\{t_1\}, s_1) \leq (\{t_1, t_2\}, s_1)$, the set *antichain* no longer is a proper antichain. □

2.4 Correct and Improved Antichain Algorithms

We first focus on solving the performance problems of Algorithms 1 and 2. Subsequently, we discuss the additional modifications that are required for Algorithm 3 to correctly decide failures-divergences refinement.

The first performance problem that we identified, *viz.*, the computational overhead induced by checking for refusal inclusion in non-stable states (which does not occur when checking for a TR-witness), can be solved in a rather straightforward manner: we only perform the check to compare the refusals of the implementation and the normal form state of the specification in case the implementation state is stable. Doing so avoids a potentially expensive search for stable states.

The second and third performance problems we identified can be solved by rearranging the computations that are conducted; these modifications are more involved. The essential observation here is that in order for the information in *antichain* to be most effective, states of the product must be added to *antichain* as soon as these are discovered, even if these have not yet been fully explored. This is achieved by maintaining, as an invariant, that $working \Subset^{\forall} antichain$ holds true; the states in *working* then, intuitively, constitute the *frontier* of the exploration. We achieve this by initialising *working* and *antichain* to consist of exactly the initial state of the product, and by extending *antichain* with all (not already discovered) successors for the state $(spec, impl)$ that is popped from *working*. As a side effect, this also resolves the third issue, as now both *working* and *antichain* are only extended with states that have not yet been discovered, *i.e.*, for which the membership test in *antichain* fails, and for which insertion of such states does not invalidate the antichain property.

The modifications we discussed above yield improved algorithms for deciding trace refinement and stable failures refinement, see the pseudocode of Algorithms 4 and 5. We postpone the discussion of their correctness until after discussing the modifications required to Algorithm 3 and its proof of correctness. The example we present below illustrates the impact of our changes.

Example 2.4.1. Consider Example 2.3.4 again, but now using Algorithm 4 to check for trace refinement. The depth-first variant of this algorithm only adds a successor state to the *working* stack once, because for every other outgoing transition it will already be part of *antichain* when it is discovered. This results in a maximum *working* stack size of at most $\mathcal{O}(1)$ entries. For each state and each successor *antichain* membership is tested once, resulting in $\mathcal{O}(n \cdot k)$ *antichain* membership tests. This is an improvement compared to the depth-first variant of Algorithm 4 of a factor $n \cdot k$ in the maximum *working* stack size and a factor k in the number of *antichain* membership tests. The bounds for the breadth-first variant are identical to the bounds for the depth-first variant, *i.e.*, maximum $\mathcal{O}(1)$ *working* queue size and $\mathcal{O}(n \cdot k)$ number of *antichain*

Algorithm 4 The improved trace refinement checking algorithm. The algorithm returns *true* iff $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in trace semantics.

```
 1: procedure REFINES-TRACE_NEW(L1, L2)
 2:     let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹_1^ε s}, ι2)
 3:     let antichain ← ∅ ⊎ ({s ∈ S1 | ι1 ⟹_1^ε s}, ι2)
 4:     while working ≠ ∅ do
 5:         pop (spec, impl) from working
 6:         for impl -a->_2 impl' do
 7:             if a = τ then
 8:                 spec' ← spec
 9:             else
10:                 spec' ← {s' ∈ S1 | ∃s ∈ spec : s ⟹_1^a s'}
11:             if spec' = ∅ then
12:                 return false
13:             if (spec', impl') ∉ antichain then
14:                 antichain ← antichain ⊎ (spec', impl')
15:                 push (spec', impl') into working
16:     return true
```

membership tests. Compared to the breadth-first variant of Algorithm 1, this is an improvement of a factor k^n in the *working* queue size and a factor k^n/n in the number of *antichain* membership tests. □

We next focus on the soundness problem of Algorithm 3. The source of the incorrectness of this algorithm can be traced back to the fact that it (partially) explores the state space of $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$, rather than $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. As illustrated by Example 2.2.15, this causes the algorithm to consider states in the product that should not be considered, thus potentially arriving at a wrong verdict. The fix to this problem is simple yet subtle, requiring a swap of the divergence tests on lines 7 and 8, and making the further exploration of the state $(spec, impl)$ conditional on the specification not diverging.

As all three algorithms presented in this section fundamentally differ (some even in the relations that they compute) from the original ones, we cannot reuse arguments for the proof of correctness presented in [137], which are based on invariants that do not hold in our case, and which rely on definitions, some of which are incomparable to ours. The correctness of our improved algorithms is claimed by the following theorem, which we repeat at the end of this section with an explicit proof.

Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs.

- REFINES-TRACE$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$.
- REFINES-STABLE-FAILURES$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$.
- REFINES-FAILURES-DIVERGENCES$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$.

Algorithm 5 The improved stable failures refinement checking algorithm. The algorithm returns *true* iff $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in stable failures semantics.

```
1:  procedure REFINES-STABLE-FAILURES_NEW(L1, L2)
2:      let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
3:      let antichain ← ∅ ⩒ ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
4:      while working ≠ ∅ do
5:          pop (spec, impl) from working
6:          if stable(impl) ∧ refusals(impl) ⊈ refusals(spec) then
7:              return false
8:          for impl -a->_2 impl' do
9:              if a = τ then
10:                 spec' ← spec
11:             else
12:                 spec' ← {s' ∈ S1 | ∃s ∈ spec : s ⟹a_1 s'}
13:             if spec' = ∅ then
14:                 return false
15:             if (spec', impl') ∉ antichain then
16:                 antichain ← antichain ⩒ (spec', impl')
17:                 push (spec', impl') into working
18:     return true
```

For the remainder of this section we fix two LTSs $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$. We focus on the proof of correctness of Algorithm 6; the correctness proofs for Algorithm 4 for deciding trace refinement and Algorithm 5 for deciding stable failures refinement proceed along the same lines.

First we show termination of Algorithm 6. To reason about the states that have been processed, we have introduced a ghost variable *done* which is initialised as the empty set (see line 4) and each pair $(spec, impl)$ that is popped from *working* at line 6 is added to *done* (line 23). For termination of the algorithm, we argue that every state in the product gets visited, and is added to *done*, at most once. A crucial observation in our reasoning is the following property of an antichain: adding elements to an antichain does not affect the membership test of elements already included. This is formalised by the lemma below.

Lemma 2.4.2. Let $(Z, \leq)$ be a partially ordered set and $\mathscr{A} \subseteq Z$ an antichain. For all elements $x, y \in Z$ if $x \Subset \mathscr{A}$ and $y \notin \mathscr{A}$ then $x \Subset (\mathscr{A} \Cup y)$ holds.

Proof. Assume arbitrary elements $x, y \in Z$ such that $x \Subset \mathscr{A}$ and $y \notin \mathscr{A}$. Recall that the definition of $\mathscr{A} \Cup y$ results in an antichain $\{z \mid z = y \vee (z \in \mathscr{A} \wedge y \not\leq z)\}$, because $y \notin \mathscr{A}$ by assumption. Consider the following two cases:

- Case $y \leq x$. Then $x \Subset (\mathscr{A} \Cup y)$ follows from the fact that $y \in (\mathscr{A} \Cup y)$.

Algorithm 6 The corrected failures-divergences refinement checking algorithm. The algorithm returns *true* iff $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ is refined by $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ in failures-divergences semantics.

```
 1: procedure REFINES-FAILURES-DIVERGENCES_NEW(𝓛1, 𝓛2)
 2:     let working be a stack containing a pair ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
 3:     let antichain ← ∅ ⩒ ({s ∈ S1 | ι1 ⟹ε_1 s}, ι2)
 4:     { done ← ∅ }
 5:     while working ≠ ∅ do
 6:         pop (spec, impl) from working
 7:         if not spec⇑ then
 8:             if impl⇑ then
 9:                 return false
10:             else
11:                 if stable(impl) ∧ refusals(impl) ⋤ refusals(spec) then
12:                     return false
13:                 for impl -a->_2 impl' do
14:                     if a = τ then
15:                         spec' ← spec
16:                     else
17:                         spec' ← {s' ∈ S1 | ∃s ∈ spec : s ⟹a_1 s'}
18:                     if spec' = ∅ then
19:                         return false
20:                     if (spec', impl') ∉ antichain then
21:                         antichain ← antichain ⩒ (spec', impl')
22:                         push (spec', impl') into working
23:         { done ← {(spec, impl)} ∪ done }
24:     return true
```

- Case $y \not\leq x$. There is an element $z \in \mathscr{A}$ such that $z \leq x$ by assumption that $x \Subset \mathscr{A}$. Because $y \not\leq x$ and $z \leq x$ we also know that $y \not\leq z$. Consequently, $z \in (\mathscr{A} \Cup y)$ and thus also $x \Subset (\mathscr{A} \Cup y)$. □

Next, we prove that *done* and *working* are disjoint, which implies that pairs present in *done* (which is a set that is easily seen to only grow) are not added to *working* again. Showing this property to be true requires two additional observations, *viz.*, (1) pairs in *working* and *done* are contained in *antichain*, and (2), *working* contains only *unique* pairs, thus representing a proper set (and, by abuse of notation, we will treat it as such). For the purpose of identifying elements in *working* we define, for a given index i, the notation $working^i$ to represent the ith pair on the stack. Now we can describe that all elements in *working* are unique by showing that $\forall i \neq j : working^i \neq working^j$ holds true. The lemma below formalises these insights.

Lemma 2.4.3. The following invariant holds in the while loop (lines 5-23) of Algo-

rithm 6:

$$(\mathit{done} \cup \mathit{working}) \Subset^{\forall} \mathit{antichain} \wedge (\forall i \neq j : \mathit{working}^i \neq \mathit{working}^j) \qquad \text{(I)}$$
$$\wedge (\mathit{done} \cap \mathit{working}) = \emptyset$$

Proof. Initially, the initial pair is added to both *working* and *antichain*, and *done* is empty, so the invariant holds trivially upon entry of the while loop.

Maintenance. At line 6 we know that $(\mathit{spec}, \mathit{impl}) \Subset \mathit{antichain}$ from $\mathit{working} \Subset^{\forall} \mathit{antichain}$. Therefore, it holds that $(\mathit{done} \cup \{(\mathit{spec}, \mathit{impl})\}) \Subset^{\forall} \mathit{antichain}$ and $(\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\}) \Subset^{\forall} \mathit{antichain}$. Furthermore, from $\forall i \neq j : \mathit{working}^i \neq \mathit{working}^j$ it follows that $(\mathit{spec}, \mathit{impl}) \notin (\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\})$. Upon executing line 6 we may therefore conclude that $((\mathit{done} \cup \{(\mathit{spec}, \mathit{impl})\}) \cap (\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\})) = \emptyset$.

Next, notice that as a result of condition $(\mathit{spec}', \mathit{impl}') \notin \mathit{antichain}$ on line 20, we have $(\mathit{spec}', \mathit{impl}') \notin (\mathit{done} \cup \mathit{working})$. Let $\mathit{working}' = \{(\mathit{spec}', \mathit{impl}')\} \cup (\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\})$ and $\mathit{done}' = \{(\mathit{spec}, \mathit{impl})\} \cup \mathit{done}$. From the fact that $(\mathit{spec}', \mathit{impl}') \notin \mathit{working}$ it follows that $\forall i \neq j : \mathit{working}'^i \neq \mathit{working}'^j$ holds true. At line 21 the $(\mathit{spec}', \mathit{impl}')$ pair is added to *antichain* and Lemma 2.4.2 ensures that predicate $(\mathit{done}' \cup \mathit{working}') \Subset^{\forall} (\mathit{antichain} \Cup (\mathit{spec}', \mathit{impl}'))$ holds. Finally, from the fact that $((\mathit{done} \cup \{(\mathit{spec}, \mathit{impl})\}) \cap (\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\})) = \emptyset$ we can also conclude that $(\mathit{done}' \cap \mathit{working}') = \emptyset$. □

Finally, we need to show that the elements in *done* and *working* are bounded by the state space of the product $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Lemma 2.4.4. The invariant $(\mathit{done} \cup \mathit{working}) \subseteq \mathsf{reachable}(\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2)$ holds in the while loop (lines 5-23) of Algorithm 6.

Proof. Initially, the pair $(\{s \in S_1 \mid \iota_1 \overset{\varepsilon}{\Longrightarrow}_1 s\}, \iota_2)$ is reachable by the empty trace because this pair is the initial state of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ by definition. Therefore, *working*, which only consists of this pair, contains pairs that are reachable as well. Moreover, *done* is empty, so the invariant holds upon entry of the while loop.

Maintenance. Let $(\mathit{spec}, \mathit{impl})$ be a pair that is popped from *working* and assume that $[\![\mathit{spec}]\!]_{\mathscr{L}_1}\Uparrow$ does not hold. Note that, by our invariant, there is a trace $\sigma \in {Act_\tau}^*$, such that $\iota \overset{\sigma}{\twoheadrightarrow} (\mathit{spec}, \mathit{impl})$. At line 13 the outgoing transition $(\mathit{impl}, a, \mathit{impl}')$ is an element of $\rightarrow_2$. Line 14 corresponds exactly to the first case of the product definition (Def. 2.2.1). Similarly, line 16 corresponds exactly to the second case of the product definition where $(\mathit{spec}, a, \mathit{spec}')$ is a transition in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1)$ because $[\![\mathit{spec}]\!]_{\mathscr{L}_1}\Uparrow$ does not hold. As such, there is a transition $(\mathit{spec}, \mathit{impl}) \xrightarrow{a} (\mathit{spec}', \mathit{impl}')$ in the product LTS. By definition of a trace and the definition of reachable this means that $(\mathit{working} \cup \{(\mathit{spec}', \mathit{impl}')\}) \subseteq \mathsf{reachable}(\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2)$. From the observation

that $(spec, impl)$ was reachable we can conclude that $(done \cup \{(spec, impl)\})$ is a subset of the reachable states as well. □

Theorem 2.4.5. Algorithm 6 terminates for finite state, finitely branching LTSs.

Proof. The inner for-loop is bounded as the number of outgoing transitions $\rightarrow_2$ is finite. The total number of state pairs in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ is finite since S_1 and S_2 are finite. From Lemma 2.4.4 it follows that *done* is a subset of the reachable state pairs. Furthermore, as $(done \cap working) = \emptyset$ by Lemma 2.4.3 we conclude that *done* strictly increases with every iteration. So, only a finite number of iterations of the while loop are possible. □

Note that these observations give an upper bound on the number of states that can be explored. Especially the absence of duplicates in *working* and the maximisation of *antichain* following from $(done \cup working) \Subset^{\forall} antichain$ do not hold for Algorithm 1, 2 and 3, as already observed in Example 2.3.4.

The remainder of this section is dedicated to proving the partial correctness of Algorithm 6, *viz.*, that when it terminates, the algorithm correctly decides failures-divergences refinement. We first revisit the claim we made in Section 2.3; before we restate and prove this claim, we prove a simplified version thereof in the next lemma.

Lemma 2.4.6. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ and actions $a \in Act_\tau$ such that $(V,s) \xrightarrow{a} (V',t)$ there is a state (U',t) such that $(U,s) \xrightarrow{a} (U',t)$ and $(U',t) \leq (V',t)$.

Proof. Let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2 = (S, \imath, \rightarrow)$ and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) = (S'_1, \imath'_1, \rightarrow'_1)$. Take any two state pairs such that $(U,s) \leq (V,s)$. Pick an arbitrary pair $(V',t) \in S$ and action $a \in Act_\tau$ such that $(V,s) \xrightarrow{a} (V',t)$. Now there are two cases to distinguish:

- Case $a = \tau$. Then a transition $s \xrightarrow{\tau}_2 t$ exists and $V = V'$. Therefore, there is also a transition $(U,s) \xrightarrow{\tau} (U',t)$ and $U = U'$. By the assumption that $(U,s) \leq (V,s)$ we know that $(U',t) \leq (V',t)$.
- Case $a \neq \tau$. Then there are transitions $V \xrightarrow{a}'_1 V'$ and $s \xrightarrow{a}_2 t$. The normalisation has, by definition, transition $V \xrightarrow{a}'_1 V'$ if and only if $V' = \{v' \in S_1 \mid \exists v \in [\![V]\!]_{\mathscr{L}_1} : v \xRightarrow{a}_1 v'\}$ and not $[\![V]\!]_{\mathscr{L}_1} \Uparrow$. Let U' be equal to $\{u' \in S_1 \mid \exists u \in [\![U]\!]_{\mathscr{L}_1} : u \xRightarrow{a}_1 u'\}$. From $[\![U]\!]_{\mathscr{L}_1} \subseteq [\![V]\!]_{\mathscr{L}_1}$ it follows that $[\![U']\!]_{\mathscr{L}_1} \subseteq [\![V']\!]_{\mathscr{L}_1}$. Furthermore, as $[\![V]\!]_{\mathscr{L}_1} \Uparrow$ does not hold it follows that not $[\![U]\!]_{\mathscr{L}_1} \Uparrow$. Therefore, $U \xrightarrow{a}'_1 U'$ exists and $(U,s) \xrightarrow{a} (U',t)$ is a transition in the product with $(U',t) \leq (V',t)$. □

We are now in a position to formally prove the claim that we made in Section 2.3. For convenience, we repeat the claim as a proposition below.

Proposition 2.4.7. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ and for every sequence $\sigma \in Act_\tau{}^*$ such that $(V,s) \xrightarrow{\sigma\!\!\!\rightarrow} (V',t)$ there is a state (U',t) such that $(U,s) \xrightarrow{\sigma\!\!\!\rightarrow} (U',t)$ and $(U',t) \leq (V',t)$.

Proof. Let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2 = (S,\iota,\rightarrow)$. The proof is by induction on the length of sequences in $Act_\tau{}^*$.

Base case. Take two pairs $(U,s),(V,s) \in S$ satisfying $(U,s) \leq (V,s)$. The empty trace can only reach $(U,s) \xrightarrow{\varepsilon\!\!\!\rightarrow} (U,s)$; similarly, we have $(V,s) \xrightarrow{\varepsilon\!\!\!\rightarrow} (V,s)$. Therefore, $(U,s) \leq (V,s)$ follows by assumption.

Inductive step. Suppose that the statement holds for all sequences in $Act_\tau{}^*$ of length i and take a sequence $\sigma \in Act_\tau{}^*$ of length i. Take arbitrary states $(V,s),(V'',r) \in S$ and action $a \in Act_\tau$ such that $(V,s) \xrightarrow{\sigma a\!\!\!\rightarrow} (V'',r)$. Then there is a state $(V',t) \in S$ such that $(V,s) \xrightarrow{\sigma\!\!\!\rightarrow} (V',t)$ and $(V',t) \xrightarrow{a} (V'',r)$. From the induction hypothesis it follows that for all $(U,s) \leq (V,s)$ there is a state $(U',t) \leq (V',t)$ such that $(U,s) \xrightarrow{\sigma\!\!\!\rightarrow} (U',t)$. By Lemma 2.4.6 and the existence of $(V',t) \xrightarrow{a} (V'',r)$ there is a state $(U'',r) \leq (V'',r)$ such that $(U',t) \xrightarrow{a} (U'',r)$. We thus conclude that $(U,s) \xrightarrow{\sigma a\!\!\!\rightarrow} (U'',r)$. □

The correctness arguments of Algorithm 6 furthermore require a lemma showing the anti-monotonicity of FD-witnesses. Such a result is needed because the antichain algorithms may explore only part of the reachable state space of a product. The anti-monotonicity property helps to show, however, that the part that *is* explored contains all relevant information.

Lemma 2.4.8. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ it holds that if (V,s) is an FD-witness then (U,s) is an FD-witness.

Proof. Take arbitrary states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ and let (V,s) be an FD-witness. It follows that $[\![V]\!]_{\mathscr{L}_1}\Uparrow$ does not hold and one of the following holds: $V = \emptyset$ or $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![V]\!]_{\mathscr{L}_1})$ or $s\Uparrow$. By monotonicity, $[\![U]\!]_{\mathscr{L}_1} \subseteq [\![V]\!]_{\mathscr{L}_1}$ implies $\mathsf{refusals}([\![U]\!]_{\mathscr{L}_1}) \subseteq \mathsf{refusals}([\![V]\!]_{\mathscr{L}_1})$, and not $[\![V]\!]_{\mathscr{L}_1}\Uparrow$ implies not $[\![U]\!]_{\mathscr{L}_1}\Uparrow$. Now, (U,s) is an FD-witness, because $[\![U]\!]_{\mathscr{L}_1}\Uparrow$ does not hold and if $V = \emptyset$ then $U = \emptyset$, or if $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![V]\!]_{\mathscr{L}_1})$ then $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}([\![U]\!]_{\mathscr{L}_1})$, or $s\Uparrow$. □

Corollary 2.4.9. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ where $(U,s) \leq (V,s)$ and for every sequence $\sigma \in Act_\tau{}^*$ it holds that if (V,s) can reach an FD-witness with σ then (U,s) can reach an FD-witness with σ as well.

Proof. Let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2 = (S,\iota,\rightarrow)$. Take arbitrary states $(U,s),(V,s) \in S$ satisfying $(U,s) \leq (V,s)$. Let (V',t) be an FD-witness and $\sigma \in Act_\tau{}^*$ a trace such

that $(V,s) \stackrel{\sigma}{\twoheadrightarrow} (V',t)$. By Lemma 2.4.7 there is a pair $(U',t) \leq (V',t)$ such that $(U,s) \stackrel{\sigma}{\twoheadrightarrow} (U',t)$. From Lemma 2.4.8 it follows that state (U',t) is an FD-witness. □

For a set of states S' of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$, let $\mathsf{FDR}(S')$ be the predicate that is true if and only if S' contains an FD-witness. For a state s in the product, we define the *distance* to a set of states S' of the product as the *shortest* distance from state s to a state in S'. If S' is unreachable, the distance is set to infinity. Formally, $\mathsf{Dist}_{S'}(s) = \min\{|\sigma| \mid \sigma \in \mathsf{traces}(\mathscr{L}) \wedge t \in S' \wedge s \stackrel{\sigma}{\twoheadrightarrow} t\}$, where $\min\{\emptyset\}$ is defined as ∞. For a set of states S'', let $\mathsf{Dist}_{S'}(S'')$ denote the shortest distance among all states in S'', formally $\mathsf{Dist}_{S'}(S'') = \min\{\mathsf{Dist}_{S'}(s) \mid s \in S''\}$. We denote the set of all reachable FD-witnesses in the product $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ by $\mathscr{F}$.

Lemma 2.4.10. For all states $(U,s),(V,s)$ of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ satisfying $(U,s) \leq (V,s)$ it holds that $\mathsf{Dist}_{\mathscr{F}}((U,s)) \leq \mathsf{Dist}_{\mathscr{F}}((V,s))$.

Proof. Let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2 = (S,\iota,\rightarrow)$. Take arbitrary states $(U,s),(V,s) \in S$ satisfying $(U,s) \leq (V,s)$. From Corollary 2.4.9 it follows that if (V,s) can reach an FD-witness by the shortest trace σ then (U,s) can also reach an FD-witness with trace σ, which by definition means that $\mathsf{Dist}_{\mathscr{F}}((U,s)) \leq \mathsf{Dist}_{\mathscr{F}}((V,s))$. □

The last lemma implies that whenever a pair is removed from the *antichain* due to an insertion of a smaller pair, the inserted (smaller) state pair has a shorter or equal distance to its closest FD-witness. This property can be used to show that the algorithm always closes in on an FD-witness during exploration and that pruning parts of the state space does not remove essential FD-witnesses from the reachable states. The latter property is captured by the following lemmas.

Lemma 2.4.11. For all states (U,s) of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ it holds that if $[\![U]\!]_{\mathscr{L}_1}\Uparrow$ then $\mathsf{Dist}_{\mathscr{F}}((U,s))$ is ∞.

Proof. Let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2 = (S,\iota,\rightarrow)$ and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) = (S'_1,\iota'_1,\rightarrow'_1)$. Take an arbitrary state $(U,s) \in S$ such that $[\![U]\!]_{\mathscr{L}_1}\Uparrow$. For any action $a \in Act_\tau$ and state $V \in S'_1$ there is no transition $U \stackrel{a}{\rightarrow}'_1 V$ by definition of $\mathsf{norm}_{\mathsf{fdr}}$. Consequently, from (U,s), only τ-transitions due to $\mathscr{L}_2$ can be taken. As a result, by definition of the product and Lemma 2.2.2, for any state $(V,t) \in S$ such that $(U,s) \stackrel{\varepsilon}{\Longrightarrow} (V,t)$ it holds that $U = V$. Thus, any reachable state (V,t) also satisfies $[\![V]\!]_{\mathscr{L}_1}\Uparrow$ and therefore cannot be an FD-witness. Hence, $\mathsf{Dist}_{\mathscr{F}}((U,s))$ is ∞. □

Lemma 2.4.12. If $\mathsf{FDR}(\mathsf{reachable}(\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2))$ is true then invariant II holds for every iteration of the while loop (lines 5-23) of Algorithm 6:

$$\mathsf{Dist}_{\mathscr{F}}(\mathit{done}) > \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) \wedge \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain}) \quad \text{(II)}$$

Proof. Assume that $\mathsf{FDR}(\mathsf{reachable}(\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2))$ holds, so there is a reachable FD-witness.

Initialisation. The set *done* is empty, so $\mathsf{Dist}_{\mathscr{F}}(\mathit{done}) = \mathsf{Dist}_{\mathscr{F}}(\emptyset) = \infty$. For *working*, which at this point only contains the initial state, the witness is reachable and therefore $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) < \infty$. The initial state is also added to *antichain*. Thus $\mathsf{Dist}_{\mathscr{F}}(\mathit{done}) > \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) \wedge \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain})$.

Maintenance. Assume that *working* is not empty and assume that $\mathsf{Dist}_{\mathscr{F}}(\mathit{done}) > \mathsf{Dist}_{\mathscr{F}}(\mathit{working})$ and $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain})$ hold. At line 6 some pair $(\mathit{spec}, \mathit{impl})$ is taken from *working*, so *working*, which by invariant I represents a set, becomes equal to $\mathit{working} \setminus \{(\mathit{spec}, \mathit{impl})\}$. Let $\mathit{done}' = \mathit{done} \cup \{(\mathit{spec}, \mathit{impl})\}$ and let $N = \mathsf{Dist}_{\mathscr{F}}((\mathit{spec}, \mathit{impl}))$. There are three cases to distinguish.

- Case $N > \mathsf{Dist}_{\mathscr{F}}(\mathit{working} \cup \{(\mathit{spec}, \mathit{impl})\})$. Removing $(\mathit{spec}, \mathit{impl})$ from *working* did not change its distance, so $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{working} \cup \{(\mathit{spec}, \mathit{impl})\})$. Because $N > \mathsf{Dist}_{\mathscr{F}}(\mathit{working})$, adding this pair to *done* yields $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) < \mathsf{Dist}_{\mathscr{F}}(\mathit{done}') \leq \mathsf{Dist}_{\mathscr{F}}(\mathit{done})$. Consider the outgoing transitions $(\mathit{impl}, a, \mathit{impl}') \in \rightarrow_2$ at line 13. The resulting pairs $(\mathit{spec}', \mathit{impl}')$ must have a distance of at least $\mathsf{Dist}_{\mathscr{F}}(\mathit{working})$, because $N - 1 \geq \mathsf{Dist}_{\mathscr{F}}(\mathit{working})$. Let $\mathit{working}' = \mathit{working} \cup \{(\mathit{spec}', \mathit{impl}')\}$. Then $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{working}')$. Let $\mathit{antichain}'$ be *antichain* if $(\mathit{spec}', \mathit{impl}')$ was not inserted and let it be $\mathit{antichain} \Cup (\mathit{spec}', \mathit{impl}')$ otherwise. By the invariant it follows that $N - 1 \geq \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain})$ and so by Lemma 2.4.10 if $(\mathit{spec}', \mathit{impl}')$ is inserted into *antichain* its distance will also not change. Therefore, $\mathsf{Dist}_{\mathscr{F}}(\mathit{done}') > \mathsf{Dist}_{\mathscr{F}}(\mathit{working}') \wedge \mathsf{Dist}_{\mathscr{F}}(\mathit{working}') = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain}')$.
- Case $0 < N \leq \mathsf{Dist}_{\mathscr{F}}(\mathit{working} \cup \{(\mathit{spec}, \mathit{impl})\})$. Observe that N must be equal to $\mathsf{Dist}_{\mathscr{F}}(\mathit{working} \cup \{(\mathit{spec}, \mathit{impl})\})$. From Lemma 2.4.11 it follows that $[\![\mathit{spec}]\!]_{\mathscr{L}_1} \Uparrow$ does not hold and so the successors of $(\mathit{spec}, \mathit{impl})$ are explored. Invariant II holds upon termination of the inner for-loop at lines 13 to 22. This follows from an invariant for the inner for-loop, which we state next.
  Let $\mathit{antichain}'$ be equal to the value of variable *antichain* after line 13 at each iteration and $\mathit{working}'$ be equal to the value of variable *working*. Furthermore, let T be the set of successors of $(\mathit{spec}, \mathit{impl})$, due to the transitions emanating from *impl*, and let done' be the successors that have been processed, *i.e.*, done' is initially empty and $(\mathit{spec}', \mathit{impl}')$ is inserted into it after line 17. It can be shown, using Lemma 2.4.10, that the following is an invariant for the inner for-loop:

$$\begin{aligned}(\mathsf{Dist}_{\mathscr{F}}(T \setminus \mathit{done}') < N \vee \mathsf{Dist}_{\mathscr{F}}(\mathit{working}') < N)\\ \wedge \mathsf{Dist}_{\mathscr{F}}(\mathit{working}') = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain}')\end{aligned}$$

  Upon termination we conclude that $(T \setminus \mathit{done}') = \emptyset$. It then follows that

$\mathsf{Dist}_{\mathscr{F}}(T \setminus \mathit{done}') = \infty$. As a consequence, we find that $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}') < N$ and therefore $\mathsf{Dist}(\mathit{working}') < \mathsf{Dist}_{\mathscr{F}}(\mathit{done} \cup \{(\mathit{spec}, \mathit{impl})\})$.

- Case $N = 0$. The state $(\mathit{spec}, \mathit{impl})$ is checked for the FD-witness conditions and the algorithm terminates. □

We conclude with the following result, which underlies the correctness of Algorithm 6.

Theorem 2.4.13. Algorithm 6 returns false if and only if an FD-witness is reachable in the product $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

Proof.

$\Longrightarrow$) Assume that Algorithm 6 returns false. This occurs when the current pair $(\mathit{spec}, \mathit{impl})$ satisfies the conditions of an FD-witness, as shown in lines 7, 8, 11 and 18 of Algorithm 6. All pairs taken from *working* are reachable according to Lemma 2.4.4, so this FD-witness is also reachable.

$\Longleftarrow$) Assume that an FD-witness is reachable in the product of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$, *i.e.*, $\mathscr{F} \neq \emptyset$. Then invariant II of Lemma 2.4.12 holds:

$$\mathsf{Dist}_{\mathscr{F}}(\mathit{done}) > \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) \wedge \mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\mathit{antichain})$$

Towards a contradiction, assume that Algorithm 6 returns true. The algorithm returns true if and only if *working* is empty, which means that $\mathsf{Dist}_{\mathscr{F}}(\mathit{working}) = \mathsf{Dist}_{\mathscr{F}}(\emptyset) = \infty$. The initial state ι of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}_1) \ltimes \mathscr{L}_2$ is equal to $(\{s \in S_1 \mid \iota_1 \stackrel{\varepsilon}{\Longrightarrow}_1 s\}, \iota_2)$ and can reach an FD-witness by assumption. Therefore, $\mathsf{Dist}_{\mathscr{F}}(\iota) < \infty$. Initially ι was inserted into *antichain* so by Lemma 2.4.2 follows that $\iota \Subset \mathit{antichain}$ and from Lemma 2.4.10 it follows that $\mathsf{Dist}_{\mathscr{F}}(\mathit{antichain}) < \infty$. Contradiction, so we conclude that if Algorithm 6 terminates then it returns false. Since termination is shown in Theorem 2.4.5 we establish that the algorithm returns false. □

We here note that analogues of Theorem 2.4.13 for Algorithms 4 and 5 can be proved along the same lines. In particular, invariants I and II, fundamental in proving termination, and proved in Lemmas 2.4.3 and 2.4.4, can be shown to hold for both algorithms using the same arguments (where, of course, the counterpart of invariant II relies on a distance to the set of TR-witnesses or SF-witnesses). Proposition 2.4.7 also holds for the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$, and Lemma 2.4.8 and Corollary 2.4.9 hold for TR-witnesses and SF-witnesses in the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Without going into these details, we here claim the correctness for Algorithms 4 and 5.

Theorem 2.4.14. Algorithm 4 returns false if and only if a TR-witness is reachable in the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$. Algorithm 5 returns false if and only if an SF-witness is reachable in the product $\mathsf{norm}(\mathscr{L}_1) \ltimes \mathscr{L}_2$.

We finish with restating the formal claim of correctness of all three improved algorithms.

Let $\mathscr{L}_1 = (S_1, \iota_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, \iota_2, \rightarrow_2)$ be two LTSs.

- REFINES-TRACE$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{tr}} \mathscr{L}_2$.
- REFINES-STABLE-FAILURES$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{sfr}} \mathscr{L}_2$.
- REFINES-FAILURES-DIVERGENCES$_{\text{NEW}}(\mathscr{L}_1, \mathscr{L}_2)$ returns true iff $\mathscr{L}_1 \sqsubseteq_{\mathsf{fdr}} \mathscr{L}_2$.

Proof. From Theorem 2.4.13 we can conclude that Algorithm 6 returns false if and only if an FD-witness is reachable. By Theorem 2.2.24 an FD-witness is only reachable if and only if $\mathscr{L}_1$ does not refine $\mathscr{L}_2$ in failures-divergences semantics. Virtually the same arguments apply for trace and stable failures refinement. □

2.5 Experimental Validation

We have conducted several experiments to compare the run time of the various algorithms to show that solving the identified issues actually improves the run time performance in practice. For this purpose we have implemented a depth-first and breadth-first variant for each of the original algorithms (Algorithms 1, 2 and 3) and improved algorithms (Algorithms 4, 5 and 6) in a branch of the mCRL2[1] toolset [23] as part of the *ltscompare* tool, which is implemented in `C++`. As the name of the tool suggests it can be used to check for various preorder and equivalence relations between labelled transition systems.

The data structures used in these implementations compute most concepts, *e.g.*, the antichain membership test and insertion, in the same way. However, the implementations of Algorithms 5 and 6 perform the check at line 6, or line 11 respectively, according to the definition of refusals we presented in Definition 2.1.5, whereas the implementations of Algorithms 2 and 3 compute the refusal check with an additional local search, according to the definition given in [137], see also Remark 2.3.2.

We first revisit Example 2.3.4 in Section 2.5.1, illustrating that the performance overhead we predict for the original algorithm for checking trace refinement also manifests itself in practice. In Section 2.5.2, we then analyse the performance of the algorithms on practical examples consisting of a model of an industrial system and models of concurrent data structures. Finally, in Section 2.5.3, we analyse the effect of using a cheap state space minimisation algorithm on the total run time of the algorithms.

All experiments and measurements have been performed on a machine with an Intel Core i7-7700HQ CPU 2.80Ghz and a 16GiB ($16 * 1024^3$ bytes) memory limit

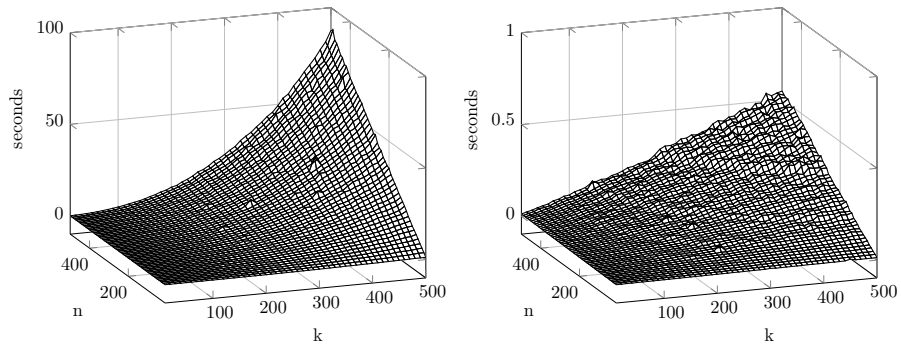
[1] www.mcrl2.org

imposed by `ulimit -Sv 16777216`. The source modifications and experiments can be obtained from the downloadable package [91].

2.5.1 Experiment I: Example 2.3.4

We have used our implementations of Algorithms 1 and 4 to measure the run time (in seconds) for checking the trace refinement $\mathscr{L}_n^k \sqsubseteq_{\mathsf{tr}} \mathscr{L}_n^k$, for all combinations of parameters $n,k \in \{10,20,\ldots,500\}$, as described in Example 2.3.4. The results of these measurements are shown as two three-dimensional plots in Figure 2.1.

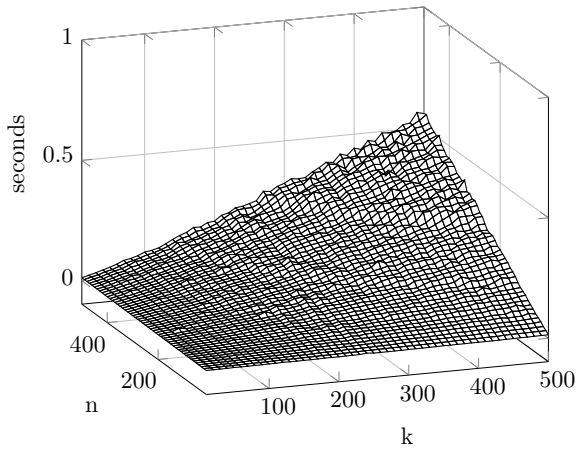
Figure 2.1: The run time results for Example 2.3.4 using the depth-first variant of Algorithm 1 on the left and our Algorithm 4 on the right.



These plots show a quadratic growth of Algorithm 1 in the parameter k and a linear growth in the parameter n. For Algorithm 4 the asymptotic growth is linear in both k and n. These observed growths coincide with the analysis that was presented in Example 2.3.4 for Algorithm 1 and on page 35 for Algorithm 4. Note that the scale of the vertical axes of both plots, displaying the run time, differs by two orders of magnitude and the highest runtime (for the $n = 500$ and $k = 500$ case) of Algorithm 1 is a factor 170 higher than that of Algorithm 4. As there is no difference in the data structures the difference in run time is entirely due to the different way of inspecting and extending *working* and *antichain*.

The breadth-first variant of Algorithm 1 was unable to complete the smallest, *i.e.*, $n = k = 10$, case within the given memory limit. However, as shown in Figure 2.2 the run time performance of the breadth-first variant of the improved algorithm is almost equivalent to its depth-first variant.

Figure 2.2: The run time for Example 2.3.4 using the breadth-first variant of Algorithm 4.



2.5.2 Experiment II: Practical Examples

The experiments that we consider are taken from two sources. First, a model of an industrial system that first exposed the performance issues in practice of a control system modelled in the Dezyne language [8] . This example is of a more traditional flavour, in which the specification is an abstract description of the behaviours at the external interface of a control system, and the implementation is a detailed model that interacts with underlying services to implement the expected interface. For reasons of confidentiality, the industrial model cannot be made available.

Second, we consider several *linearisability tests* of concurrent data structures. These models have been taken from [112], and consist of six implementations of concurrent data types that, when *trace refining* their specifications, are guaranteed to be linearisable. As in [137], we approximate trace refinement by the stronger stable failures refinement. For these models, the implementation and specification pairs are based on the same descriptions; the difference between the two is that the specification uses a simple construct to guarantee that each method of the concurrent data structure executes atomically. This significantly reduces the non-determinism and the number of transitions in the specification models.

In Table 2.1 the origin of each model, the number of states and transitions of

each implementation and specification LTS, and whether the stable failures refinement relation holds is shown.

Table 2.1: The number of states and transitions in each benchmark.

Model	Ref.	states spec	trans. spec	$\sqsubseteq_{\mathsf{sfr}}$	states impl	trans. impl
Industrial	-	24	45	True	24 551	45 447
Coarse set	[74]	50 488	64 729	True	55 444	145 043
Fine-grained set	[74]	3 720	3 305	True	5 077	9 006
Lazy set	[74]	3 565	3 980	True	24 496	41 431
Optimistic set	[74]	25 435	28 154	True	234 332	389 344
Non-blocking queue	[125]	1 248	1 473	False	3 030	5 799
Treiber stack	[134]	87 389	124 740	True	205 634	564 862

The run time measurements of both Algorithms 2 and 5 with both the depth-first and breadth-first variants is shown in Table 2.2. The run times that we report are the averages obtained from five consecutive runs.

Table 2.2: Run time comparison between Algorithm 2 and Algorithm 5 using depth-first (df) and breadth-first (bf) exploration.

Model	Alg. 2 df (s)	Alg. 2 bf (s)	Alg. 5 df (s)	Alg. 5 bf (s)
Industrial	1.36	296.29	0.15	0.17
Coarse set	9.15	†	8.61	9.06
Fine-grained set	0.37	†	0.32	0.46
Lazy set	1.19	†	1.02	1.26
Optimistic set	16.96	†	14.13	22.67
Non-blocking queue	0.03	0.17	0.02	0.09
Treiber stack	148.39	†	137.52	352.59

Here, we observe that the depth-first variant of both algorithms perform similarly with a small run time advantage for Algorithm 5. However, for the breadth-first variants our algorithm is able to complete all experiments, whereas Algorithm 2 reaches the memory limit, indicated by †, in five cases and only completes two cases successfully.

To gain more insight into the performance differences between both algorithms we repeat the experiments and report a number of performance metrics. The reported metrics are the maximum *working* size and the number of *antichain* membership test that fail (misses), succeed (hits) and the maximum *antichain* size during the exploration. We report the maximum size instead of its size upon termination as these

do not necessarily coincide, because inserting an element can evict one or more pairs in *antichain*. The following two tables (Tables 2.3 and 2.4) show the discussed metrics for the depth-first variant of both algorithms.

Table 2.3: Performance metrics for the depth-first variant of Algorithm 2.

Model	*working* max	*antichain* hits	*antichain* misses	*antichain* max
Industrial	74	36 544	43 419	43 091
Coarse set	96	93 330	58 438	55 444
Fine-grained set	60	5 786	7 575	5 077
Lazy set	61	21 184	30 771	24 496
Optimistic set	96	234 692	354 068	238 726
Non-blocking queue	52	548	672	591
Treiber stack	101	1 238 727	756 692	234 118

Table 2.4: Performance metrics for the depth-first variant of Algorithm 5.

Model	*working* max	*antichain* hits	*antichain* misses	*antichain* max
Industrial	69	36 369	43 090	43 091
Coarse set	96	93 330	58 438	55 444
Fine-grained set	60	5 786	7 575	5 077
Lazy set	61	21 184	30 771	24 496
Optimistic set	96	234 692	354 068	238 728
Non-blocking queue	43	520	641	634
Treiber stack	101	1 238 727	756 692	234 119

We observe in Tables 2.3 and 2.4 that only for the industrial and non-blocking queue models the performance metrics are different. An explanation for this is that because the antichain membership test is delayed (in Algorithm 2), more pairs are added to *working* and these additional pairs increase the number of *antichain* checks. In all other cases, the difference in run time can only be the result of the different refusal computation implementation, as the number of *antichain* operations is the same.

The following two tables (Tables 2.5 and 2.6) show the obtained performance metrics for the breadth-first variants of both algorithms. In the experiments where the refinement checking terminates early, due to reaching the memory limit, we report the last observed measurements.

Table 2.5: Performance indicators for the breadth-first variant of Algorithm 2.

Model	*working* max	*antichain* hits	*antichain* misses	*antichain* max
Industrial	549 263	5 459 028	12 888 388	43 091
Coarse set	4 710 289	13 870	7 807 403	3 629
Fine-grained set	6 604 516	180 669	15 547 890	1 900
Lazy set	6 726 497	130 523	14 852 835	4 306
Optimistic set	6 366 524	38 649	14 238 042	4 439
Non-blocking queue	6 262	3 078	14 560	274
Treiber stack	5 829 902	76 114	8 340 606	4 811

Table 2.6: Performance indicators for the breadth-first variant of Algorithm 5.

Model	*working* max	*antichain* hits	*antichain* misses	*antichain* max
Industrial	2 243	36 369	43 090	43 091
Coarse set	3 411	96 167	60 332	55 444
Fine-grained set	434	7 192	9 657	5 077
Lazy set	1 748	24 340	35 192	24 496
Optimistic set	15 209	292 525	434 218	234 352
Non-blocking queue	338	3 426	4 032	2 675
Treiber stack	139 218	2 411 614	1 523 830	214 795

From these results it is clear to see that for the breadth-first variant of Algorithm 2, delaying the *antichain* insertion of discovered state pairs results in an enormous overhead. The size of the *antichain* remains quite small, which causes many discovered pairs to fail the antichain membership test. As each pair that fails the membership test is added to *working*, it causes the *working* queue to grow rapidly, until it reaches the memory limit. On the other hand, for Algorithm 5 we can observe that the number of successful (and unsuccessful) *antichain* membership test is quite similar to its depth-first variant. There can be some differences between these variants as the pairs are discovered in a different order. The increase of the *working* size has the same reason as for ordinary breadth-first search, which depends on the out degree of the visited pairs.

To verify that the difference in performance of the depth-first variants is due to the changes of the refusal computation we have implemented another variant of Algorithm 2 with the stability check of *impl* added. The run time impact of this change for both depth-first and breadth-first variants of Algorithm 2 is shown in Table 2.7.

As expected, the run time for this alternative depth-first variant closely matches the run time of the depth-first variant of the improved algorithm. The alternative

Table 2.7: Run time results for Algorithm 2 with the stability check of *impl*.

Model	Alg. 2 df (s)	Alg. 2 bf (s)
Industrial	0.17	26.32
Coarse set	9.59	†
Fine-grained set	0.36	†
Lazy set	1.12	†
Optimistic set	15.85	†
Non-blocking queue	0.03	†
Treiber stack	156.67	†

breadth-first variant of Algorithm 2 is still not able to complete most experiments, but the industrial case has improved quite significantly. However, the non-blocking queue experiment now reaches the set memory limit. For this we provide the following explanation. Note that in case of a failing refinement the exploration stops when a suitable (SF-)witness has been found, which must exist as stable failures refinement does not hold. Recall that the computation of refusals for (possibly) unstable states as defined in [137], see Remark 2.3.2, has been implemented using a separate local search for stable states. We think that in the previous case such an SF-witness was found for an unstable state using this local search. However, in the alternative version the algorithm continues the exploration of the LTSs when encountering an unstable state (in $\mathscr{L}_2$), which causes the *working* queue to reach the memory limit.

Finally, we repeat the same experiments while checking for failures-divergences refinement. This has only been done for Algorithm 6 as the original algorithm for failures-divergences refinement is incorrect. The run time measurements and the expected result of the failures-divergences refinement check are presented in Table 2.8.

Table 2.8: The run time results for checking failures-divergences refinement using Algorithm 6.

Model	Alg. 6 df (s)	Alg. 6 bf (s)	$\sqsubseteq_{\mathsf{fdr}}$
Industrial	0.05	0.05	False
Coarse set	8.68	9.29	True
Fine-grained set	0.33	0.48	True
Lazy set	1.04	1.33	True
Optimistic set	14.55	23.81	True
Non-blocking queue	0.08	0.10	True
Treiber stack	140.70	363.34	True

The run time results of Table 2.8 show that deciding failures-divergences refinement has a similar performance to deciding stable failures.

2.5.3 State Space Minimisation as Preprocessing

The size of the transition systems has a major impact on the practical run time of the refinement checking algorithms we studied, as can also be seen from, *e.g.*, Tables 2.1 and 2.2. Note that this is particularly true of the size of the specification LTS, whose normal form can be exponentially larger than the specification itself. As an alternative to the pruning achieved using antichains, reducing the size of the specification as a preprocessing step to checking for refinement may therefore be an effective tool in improving on the practical run time of these algorithms. Of course, it is desirable that the computational overhead of the reduction remains minimal. One possibility is to minimise transition systems using one of the many equivalence relations available for labelled transition systems, see, *e.g.* [60, 13]. When choosing such an equivalence it is important that it has the property that, apart from an appealing run time complexity, the observations, *i.e.*, $\mathsf{weaktraces}$, $\mathsf{failures}$ and $\mathsf{divergences}$, that are extracted from equivalent states are the same.

Strong bisimilarity is known to preserve the $\mathsf{weaktraces}$, $\mathsf{failures}$ and $\mathsf{divergences}$ observations of equivalent states. However, a more substantial state space reduction can often be achieved by considering equivalences that treat the special action τ as invisible, as the given LTSs often contain τ-transitions. In [122], Roscoe suggests to use a variant of weak bisimulation, *viz.*, *divergence respecting weak bisimulation*, to minimise a transition system. For *divergence respecting weak* bisimulation it is known that it is a suitable abstraction; see the following theorem [122, Theorem 9.2].

Theorem 2.5.1. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS. For two states $s, t \in S$ that are divergence respecting weak bisimilar it holds that $\mathsf{weaktraces}(s) = \mathsf{weaktraces}(t)$, $\mathsf{divergences}(s) = \mathsf{divergences}(t)$, $\mathsf{failures}(s) = \mathsf{failures}(t)$. Hence, also $\mathsf{failures}_\perp(s) = \mathsf{failures}_\perp(t)$.

From a computational point of view, however, (divergence respecting) weak bisimulation is not particularly promising. For instance, the best known algorithm [116] for computing weak bisimulation has a worst-case time complexity of $\mathcal{O}(m \cdot n)$, where n is the number of states and m the number of transitions. Such run time complexities are non-neglible and may result in an undesirable overhead.

In practice, divergence-respecting weak bisimulation often coincides with *divergence preserving branching bisimulation* and it has the far more appealing worst-case run time complexity of $\mathcal{O}(m \cdot \log n)$ [83], which is equivalent to the run time complexity of computing strong bisimulation [110]. Furthermore, the implementation for divergence preserving branching bisimulation is also far more efficient than the

one for divergence-respecting weak bisimulation in the mCRL2 toolset. Divergence-preserving branching bisimulation is stronger than divergence respecting weak bisimulation, *i.e.*, two states that are divergence-preserving branching bisimilar are also divergence respecting weak bisimilar. Divergence-preserving branching bisimilarity is, by Theorem 2.5.1, therefore a suitable abstraction.

Remark 2.5.2. Note that Theorem 6.1 of [57] states that when comparing two systems of which one contains no τ-transitions, weak and branching bisimilarity coincides. It is argued that this often applies when comparing specifications with implementations. This trivially generalises to divergence-respecting weak and divergence preserving branching bisimilarity. Also in case of our benchmarks, both relations yield only a minimal difference in size for all models, with the exception of the industrial implementation model. For that model, however, the costs of computing the weak-bisimulation reduction far exceeds the costs of performing the refinement check. Perhaps this could be partially mitigated by more efficient (divergence respecting) weak bisimulation algorithms, but that has not been further investigated.

The algorithm that decides divergence-preserving branching bisimulation equivalence between two states can also be adapted to a minimisation procedure with the same $\mathcal{O}(m \cdot \log n)$ time complexity. We have made the preprocessing step of minimisation modulo divergence-preserving branching bisimulation available as an option in our tool. In Table 2.9 the number of states and transitions of each model of Section 2.5.2, *after* minimisation modulo divergence-preserving branching bisimulation, and whether the specification and implementation LTSs are divergence-preserving branching bisimilar is shown.

Table 2.9: The number of states and transitions after diverging preserving branching bisimulation minimisation and whether the LTSs are equivalent in divergence-preserving branching bisimulation semantics denoted by $\leftrightarrows_{db}$.

Model	states spec	trans. spec	$\leftrightarrows_{db}$	states impl	trans. impl
Industrial	24	45	False	4 626	14 380
Coarse set	1 089	3 618	True	1 089	3 618
Fine-grained set	92	210	True	92	210
Lazy set	92	210	True	92	210
Optimistic set	170	410	True	170	410
Non-blocking queue	119	274	False	163	378
Treiber stack	7 988	26 070	True	7 988	26 070

Observe that for most of the models, the minimised implementation and specification LTSs are of equal size; indeed, in those cases the implementation and specification

are divergence-preserving branching bisimulation equivalent, so no further stable failures refinement check would be needed.

One option would therefore be to apply the minimisation to both implementation and specification LTSs. This approach turns out to be beneficial for the Treiber stack example, obtaining a run time of 3 seconds to determine stable failures refinement. The approach is not beneficial for the other examples. Moreover, minimising the implementation might even be less effective in case the refinement relation between specification and implementation does not hold, in which case the refinement check will probably quickly determine this fact.

We therefore measure the effect of using minimised specifications, but unmodified implementations. The run time measurements of checking stable failures refinement using Algorithms 2 and 5 using the minimised specification LTS is shown in Table 2.10. The time that it takes to compute the divergence-preserving branching bisimulation minimisation is presented in the last column and the other measurements are the run time of the algorithm including preprocessing.

Table 2.10: Run time comparison between the original algorithm (Algorithm 2) and the improved algorithm (Algorithm 5) using depth-first (df) and breadth-first (bf) exploration where the specification is reduced modulo divergence-preserving branching bisimulation.

Model	Alg. 2 df (s)	Alg. 2 bf (s)	Alg. 5 df (s)	Alg. 5 bf (s)	Reduction (s)
Industrial	1.38	293.10	0.16	0.17	0.01
Coarse set	0.74	†	0.69	0.69	0.10
Fine-grained set	0.04	†	0.04	0.04	0.01
Lazy set	0.21	†	0.15	0.15	0.01
Optimistic set	2.52	†	1.59	1.57	0.04
Non-blocking queue	0.02	0.04	0.02	0.02	0.01
Treiber stack	8.19	†	6.61	11.71	0.24

Comparing these results with Table 2.2 shows that reducing the specification modulo divergence-preserving branching bisimulation can indeed substantially improve the performance of the antichain-based algorithms. In particular, it never degrades the performance of our algorithms as the preprocessing time is negligible. For failures-divergences refinement the results, using Algorithm 6, are similar, as is shown in Table 2.11.

Table 2.11: The run time results for checking failures-divergences refinement using Algorithm 6 where the specification is reduced module divergence-preserving branching bisimulation.

Model	Alg. 6 df (s)	Alg. 6 bf (s)
Industrial	0.05	0.06
Coarse set	0.75	0.70
Fine-grained set	0.04	0.04
Lazy set	0.15	0.15
Optimistic set	1.61	1.70
Non-blocking queue	0.02	0.02
Treiber stack	6.76	12.13

2.6 Conclusion

Our study of the antichain-based algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement presented in [137] revealed that the failures-divergences refinement algorithm is incorrect. All three algorithms perform suboptimally when implemented using a depth-first search strategy and poorly when implemented using a breadth-first search strategy. Furthermore, all three algorithms violate the claimed antichain property. We propose alternative algorithms for which we have shown correctness and which utilise proper antichains. Our experiments indicate significant performance improvements for deciding trace refinement, stable failures refinement and a performance of deciding failures-divergences refinement that is comparable to deciding stable failures refinement. We also show that preprocessing using divergence-preserving branching bisimulation offers substantial performance benefits. The implementation of our algorithms is available in the open source toolset mCRL2 [23] and is currently used as the backbone in the commercial F-MDE toolset Dezyne; see also [8].

In this work we have only focussed on comparing the performance of the erroneous antichain-based algorithms to the corrected algorithms. However, an interesting comparison would be between the corrected antichain-based algorithms and the classical refinement checking algorithms based on normalisation of the specification for a wide variety of (practical) specifications. The classical algorithm either normalise the specification completely before checking refinement, or normalise the specification on-the-fly. Such a comparison could be useful in order to decide which algorithm should be applied when. Similarly, different statespace reduction algorithms could be compared within the mCRL2 toolset. However, currently such a comparison might not be useful since the mCRL2 toolset implements for example weak bisimulation

using a preprocessing step followed by branching bisimulation, which might not be the most efficient implementation. Finally, our implementation could be compared to the state-of-the-art algorithms implemented in the FDR [54] toolset, but this has proven to be difficult due to the different input languages that these toolsets have.

In terms of theoretical contributions an interesting route would be to see if antichains can also be employed in other refinement algorithms, for example to algorithms used to decide so-called fair testing refinement [119]. Initial findings suggest that this might be possible, but further research in this direction would be needed to obtain correct and efficient refinement algorithms for fair testing.

Chapter 3

Decomposing Monolithic Processes in a Process Algebra with Multi-actions

The mCRL2 language [63] is a process algebra that can be used to specify the behaviour of communicating processes with data parameters. It has the usual ACP-style operators for modelling non-deterministic choice, sequential composition, parallel composition and recursion. A powerful yet somewhat unconventional language construct of mCRL2 is the *multi-action*, which allows for specifying that atomic actions can happen simultaneously.

Specifications written in mCRL2 can be analysed using the corresponding mCRL2 toolset [23]. The mCRL2 toolset [23] translates a process specification to an equivalent monolithic recursive process, replacing all interleaving parallelism by non-determinism, action prefixing and recursion.

Translating a complicated process specification into a simpler normal form, in this case the monolithic process, has several advantages due to the fact that we only have to deal with a simple structure instead of the full process algebra in all subsequent analyses. First of all, the design and implementation of state space exploration algorithms can be greatly simplified. Furthermore, the design of effective static analysis techniques on the global behaviour of the specification is also easier.

Such static analysis techniques can help to reduce the *size* of the state space underlying the monolithic process, where the size is the sum of the number of states and transitions. One example is a static analysis to detect live variables as presented in [114], where variables are live whenever their values influence the behaviour. Other

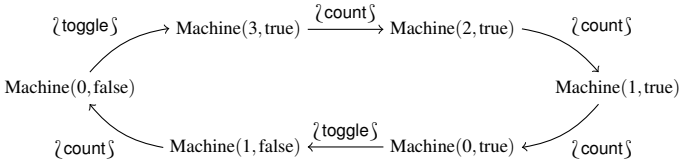


Figure 3.1: Behaviour of a machine that alternates between two modes of operation. The transitions are labelled with multi-sets, indicated as ⟅ ⟆, of actions.

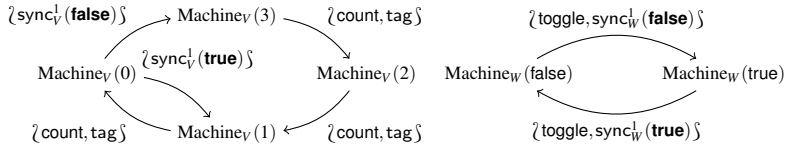


Figure 3.2: Behaviour of the decomposition processes.

static analysis are the elimination of constant parameters as presented in [67]. However, the static analysis techniques available at the moment are not always strong enough to mitigate the state space explosion problem for this monolithic process even though its state space can often be minimised with respect to some equivalence relation after state space exploration.

In this chapter, we define a decomposition technique (which we refer to as a *cleave*) of a monolithic process. Our technique takes as input such a process and a partitioning of its data parameters, and it produces two new processes. To illustrate the idea, consider a machine that alternates between two modes, where switching modes has a certain delay. The behaviour of this machine is modelled by the labelled transition system in Figure 3.1. Assume that this machine is described by a single recursive mCRL2 process with two parameters: a natural number representing the counter for the delay and a Boolean for representing the mode of the machine. Using the partition that 'splits' these two parameters, our technique will decompose this machine into two recursive processes (*components*) with their respective behaviour shown in Figure 3.2. Observe that indeed the states of a component rely on only one of the two parameters. Note that the transition systems of both components include sync and tag actions that do not occur in the transition system of the original machine. These are generated by our technique and are needed to model the *interface* between the two components such that under a suitable synchronisation context the parallel composition of these components *is* equivalent (*strongly bisimilar*) to the original monolithic process.

Decomposing a monolithic process may help to partly sidestep the *state space explosion* that is due to the interleaving of parallel processes that is encoded in the monolithic process. This follows from the observation that the state spaces of the components resulting from a decomposition can be (much) smaller than the state space of the monolithic process; these may therefore be easier to obtain. By first minimising the state spaces of these components with respect to bisimilarity *before* computing their composition, also the state space of the latter remains (much) smaller than that of the monolithic process. Since strong bisimilarity is a congruence for all operators of mCRL2, the resulting state space is still strongly bisimilar to that of the monolithic process, meaning that no information is lost.

Theoretically, the main challenge in defining a decomposition technique is to ensure that it results in components that, when combined appropriately, behave indistinguishably from the monolithic process from which they were derived. This is the problem of finding a *valid decomposition*. We illustrate that there may be multiple valid decompositions of a monolithic process. The main practical challenge is therefore to identify a universally applicable decomposition technique that yields valid decompositions, and which is capable of sidestepping the state space explosion problem. In this chapter, we show that the techniques that we develop can, to a large extent, be automated. Moreover, we provide detailed proofs of the main results and include an extensive set of experiments.

Summarising the contributions of our work are as follows:

- we formalise the notion of a decomposition (see Section 3.2) and the notion of validity of a decomposition,
- we present a generally applicable decomposition technique and provide sufficient conditions for this decomposition to be valid (see Section 3.3),
- we show that *state invariants* [68] can be used to obtain even smaller state spaces by restricting the interfaces of the components resulting from the decomposition (Section 3.4),
- we provide algorithms that, based on a user-defined selection of process parameters, extract two components from a monolithic process (see Section 3.5),
- we confirm the practical applicability of our techniques on several cases in (see Section 3.6).

Related Work Several different techniques are related to this type of decomposition. Most notably, the work on decomposing Petri nets into a set of automata [14] also aims to speed up state space exploration by means of decomposition. However, Petri nets have a clear structure and lack (possibly complex) data expressions that must be taken into account. The work on functional decomposition [18] describes a technique to decompose a specification based on a partitioning of the action labels for a basic

fragment of LOTOS, where processes have no data parameters. In [85] it was shown how this type of decomposition can be achieved for mCRL2 processes. These works use the structure of the original process specification to perform a decomposition with the intention to distribute the components, and are not necessarily interested in compositional verification techniques and it is not clear whether reductions can be achieved in this way. Furthermore, a decomposition technique was used in [68] to improve the efficiency of equivalence checking. However, that work considers processes that are already in a parallel composition and further decomposes them based on the actions that occur in each component.

Decompositional minimisation is also related to *compositional minimisation*, in which the objective is to replace the state space of each component in a (given) parallel composition by an equivalent, smaller state space, while preserving the behaviour of the original specification [130, 131]. A problem that is common to compositional minimisation and decompositional minimisation is that the size of the state spaces belonging to individual components summed together might still exceed the size of the original state space [51]. One way to (partly) avoid this is by specifying *interface constraints* (also known as *environmental constraints* or *context constraints*), see [62, 28]. The state invariants in our work serve a similar purpose, but the mechanism is different since interface constraints are action-based whereas invariants are state-based. Another possibility is to find a more suitable order in which components are explored and minimised, since the order heavily influences the size of the intermediate state spaces. Heuristics for determining this order can be very effective in practice [34]; such heuristics are also relevant for the application of our decomposition technique.

One advantage of the decomposition technique over compositional minimisation is that our interfaces can be derived from the conditions present in the monolithic process. These interfaces can also be further strengthened with state invariants. Secondly, the components resulting from the decomposition are not limited to the user-defined processes present in the specification. Our decomposition technique is thus more flexible, and may yield more optimal compositions. Indeed, the case studies on which we report support both observations.

3.1 Preliminaries

We assume the existence of an abstract data theory that describes data sorts, where sorts are sometimes referred to as types in other contexts. Each sort D has an associated non-empty semantic domain denoted by $\mathbb{D}$. The existence of sorts *Bool* and *Nat* with their associated Boolean ($\mathbb{B}$) and natural number ($\mathbb{N}$) semantic domains respectively, with standard operators is assumed. Furthermore, we assume the existence of an infinite set of *sorted variables*. We use $e : D$ to indicate that e is an expression (or

variable) of sort D. The set of free variables of an expression e is denoted $\mathsf{FV}(e)$, and a variable that is not free is called *bound*. An expression e is *closed* iff $\mathsf{FV}(e) = \emptyset$. A substitution σ is a total function from variables to closed data expressions of their corresponding sort. For a substitution σ, we write $\sigma[x \leftarrow e]$ to denote the substitution σ' such that $\sigma'(x) = e$ and for all $y \neq x$, we have $\sigma'(y) = \sigma(y)$. We use $\sigma(e)$ to denote the syntactic replacement of variables in expression e by their substituted expression.

An *interpretation* function, denoted by $[\![\ldots]\!]$, maps syntactic objects to values within their corresponding semantic domain. We assume that $[\![e]\!]$ for closed expressions e is already defined. Semantic objects are typeset in *boldface* to differentiate them from syntax, *e.g.*, the semantics of expression $1+1$ is $\mathbf{2}$. We denote *data equivalence* by $e \approx f$, which is true iff $[\![e]\!] = [\![f]\!]$; for other operators we use the same symbol in both syntactic and semantic domains. Data equivalence is lifted to equivalence of substitutions in the usual way, *i.e.*, $\sigma \approx \sigma'$ iff for all variables x we have $\sigma(x) \approx \sigma'(x)$. We adopt the usual principle of substitutivity; *i.e.*, for all substitutions σ, σ' and expressions e it holds that if $\sigma \approx \sigma'$ then $\sigma(e) \approx \sigma'(e)$.

We denote a *vector* of length $n+1$ by $\vec{d} = \langle d_0, \ldots, d_n \rangle$. The empty vector is denoted by $\langle\rangle$. Two vectors are equivalent, denoted by $\langle d_0, \ldots, d_n \rangle \approx \langle e_0, \ldots, e_n \rangle$, iff their elements are *pairwise equivalent*, *i.e.*, $d_i \approx e_i$ for all $0 \leq i \leq n$. Given a vector $\langle d_0, \ldots, d_n \rangle$ and a subset $I \subseteq \mathbb{N}$, we define the *projection*, denoted by $\langle d_0, \ldots, d_n \rangle_{|I}$, as the vector $\langle d_{i_0}, \ldots, d_{i_l} \rangle$ for the largest $l \in \mathbb{N}$ such that $i_0 < i_1 < \ldots < i_l \leq n$ and $i_k \in I$ for $0 \leq k \leq l$. For a vector $\langle d_0 : D_0, \ldots, d_n : D_n \rangle$ we write $\vec{d} : \vec{D}$ and denote the projection for a subset of indices $I \subseteq \mathbb{N}$ by $\vec{d}_{|I} : \vec{D}_{|I}$. Finally, we define $\mathsf{Vars}(\vec{d}) = \{d_0, \ldots, d_n\}$.

A *multi-set* over a set A is a total function $m : A \to \mathbb{N}$; we refer to $m(a)$ as the *multiplicity* of a and we write $\wr \ldots \wr$ for a multi-set where the multiplicity of each element is either written next to it or omitted when it is one. For instance, $\wr a : 2, b \wr$ has elements a and b with multiplicity two and one respectively, and all other elements have multiplicity zero. For multi-sets $m, m' : A \to \mathbb{N}$, we write $m \subseteq m'$ iff $m(a) \leq m'(a)$ for all $a \in A$. Multi-sets $m + m'$ and $m - m'$ are defined pointwise: $(m + m')(a) = m(a) + m'(a)$ and $(m - m')(a) = \max(m(a) - m'(a), 0)$ for all $a \in A$.

3.1.1 Labelled Transition Systems

Let Λ be the set of (sorted) action *labels*. We use D_a to indicate the sort of action label $a \in \Lambda$. The set of all multi-sets over $\{a(\mathbf{e}) \mid a \in \Lambda, \mathbf{e} \in \mathbb{D}_a\}$ is denoted Ω. Note that $\mathbb{D}_a$ is the semantic domain of D_a. In examples we often omit the expression and parentheses whenever $\mathbb{D}_a$ consists of a single element.

Definition 3.1.1. A labelled transition system with multi-actions, abbreviated LTS, is a tuple $\mathscr{L} = (S, Act, \rightarrow)$ where S is a set of states; $Act \subseteq \Omega$ and $\rightarrow \subseteq S \times Act \times S$ is a labelled transition relation.

We typically use ω to denote an element of Act and we write $s \xrightarrow{\omega} t$ whenever $(s,\omega,t) \in \rightarrow$. As usual, a finite LTS can be depicted as an edge-labelled directed graph, where vertices represent states, the labelled edges represent the transitions. The left graph of Figure 3.2 (see page 58) depicts an LTS with four states and five transitions, which are labelled with multi-actions $\{\!|\mathsf{count},\mathsf{tag}|\!\}$, $\{\!|\mathsf{sync}_V^1(\mathbf{true})|\!\}$ and $\{\!|\mathsf{sync}_V^1(\mathbf{false})|\!\}$. The size of a labelled transition system is given by the number of states and transitions combined.

We recall the well-known strong bisimulation equivalence relation on states of an LTS [104].

Definition 3.1.2. Let $\mathscr{L} = (S, Act, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is a *(strong) bisimulation relation* iff for all $s\,R\,t$:

- if $s \xrightarrow{\omega} s'$ then there is a state $t' \in S$ such that $t \xrightarrow{\omega} t'$ and $s'\,R\,t'$, and
- if $t \xrightarrow{\omega} t'$ then there is a state $s' \in S$ such that $s \xrightarrow{\omega} s'$ and $s'\,R\,t'$.

States s and t are *bisimilar*, denoted $s \leftrightarroweq t$, iff $s\,R\,t$ for a bisimulation relation R.

3.1.2 Linear Process Equations

We draw inspiration from the process algebra mCRL2 [63] to describe the elements of an LTS; similar language concepts and constructs may appear in other shapes elsewhere; for example ACP [6], CCS [103] and SCCS [104].

Definition 3.1.3. *Multi-actions* are defined as follows:

$$\alpha \;::=\; \tau \;\mid\; a(e) \;\mid\; \alpha|\alpha$$

Constant τ represents the empty multi-action and $a \in \Lambda$ is an action label with an expression e of sort D_a. The semantics of a multi-action α, given a substitution σ, is denoted by $[\![\alpha]\!]_\sigma$ and is an element of Ω. It is defined inductively as follows: $[\![\tau]\!]_\sigma = \{\!|\,|\!\}$, $[\![a(e)]\!]_\sigma = \{\!|a([\![\sigma(e)]\!])|\!\}$ and $[\![\alpha|\beta]\!]_\sigma = [\![\alpha]\!]_\sigma + [\![\beta]\!]_\sigma$. If α is a closed expression then the substitution is typically omitted.

Example 3.1.4. Consider the multi-action $\mathsf{toggle}|\mathsf{sync}_W^1(\mathsf{false})$. Since this is a closed expression, the semantics $\{\!|\mathsf{toggle},\mathsf{sync}_W^1(\mathbf{false})|\!\}$ of the multi-action is independent of a substitution. The semantics of the multi-action $\mathsf{toggle}|\mathsf{sync}_W^1(x)$, in the context of substitution σ satisfying $\sigma(x) = \mathsf{true}$ is $\{\!|\mathsf{toggle},\mathsf{sync}_W^1(\mathbf{true})|\!\}$.

The states and transitions of an LTS are described by means of monolithic processes called *linear process equations*, which consist of a number of *condition-action-effect* statements, referred to as *summands*. Each summand symbolically represents a partial transition relation between the current and the next state for a multi-set of action labels. Let PN be a set of (sorted) *process names*.

Definition 3.1.5. A *linear process equation* (LPE) is an equation of the form:

$$P(d:D) = \sum_{e_0:E_0} c_0 \to \alpha_0 \,.\, P(g_0) + \ \ldots\ + \sum_{e_n:E_n} c_n \to \alpha_n \,.\, P(g_n)$$

Where $P \in PN$ is the *process name*, d is the *process parameter*, and each:

- E_i is a sort ranged over by *sum* variable e_i (where $e_i \neq d$),
- c_i is the *enabling condition*, a boolean expression so that $\mathsf{FV}(c_i) \subseteq \{d, e_i\}$,
- α_i is a multi-action τ or $a_i^1(f_i^1)|\ldots|a_i^{n_i}(f_i^{n_i})$ such that each $a_i^k \in \Lambda$ and f_i^k is an expression of sort $D_{a_i^k}$ such that $\mathsf{FV}(f_i^k) \subseteq \{d, e_i\}$,
- g_i is an *update* expression of sort D, satisfying $\mathsf{FV}(g_i) \subseteq \{d, e_i\}$.

The $+$-operator denotes a non-deterministic choice among the summands of a given LPE; the $\sum$-operator describes a non-deterministic choice among the possible values of the associated sum variable bound by the $\sum$-operator. We omit the $\sum$-operator when the sum variable does not occur freely within the condition, action and update expressions. We use $\mathop{+}_{i\in I}$ for a finite set of *indices* $I \subseteq \mathbb{N}$ as a shorthand for a number of summands.

Note that similar structures also occur elsewhere. For example in Extended Finite State Machines [27] or Symbolic Transition Graphs [72], process parameters are sometimes called *state variables* and summands are referred to as *transitions*.

We often consider LPEs where the parameter sort D represents a *vector*; in that case we write $d_0 : D_0, \ldots, d_n : D_n$ to indicate that there are $n+1$ parameters where parameter d_i has sort D_i. Similarly, we also generalise the action sorts and the sum operator in LPEs, where we permit ourselves to write $a(f_0, \ldots, f_k)$ and $\sum_{e_0:E_0,\ldots,e_l:E_l}$, respectively.

Let P be the set of expressions $P(\iota)$, where $P \in PN$ and ι is a closed expression of sort D (the sort of P). The labelled transition system induced by a (set of) LPE(s) is then formally defined as follows.

Definition 3.1.6. The operational semantics associated with expressions of P is the LTS $(\mathsf{P}, \Omega, \to)$, where the transition relation $\to$ is defined as the smallest relation obtained as follows: for each LPE $P(d:D) = \mathop{+}_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(g_i)$ and for all indices $i \in I$, closed expressions $\iota : D$ and substitutions σ such that $\sigma(d) = \iota$ there is a transition $P(\iota) \xrightarrow{[\![\sigma(\alpha_i)]\!]} P(\sigma(g_i))$ iff $[\![\sigma(c_i)]\!] = \mathbf{true}$.

For a given expression $P(\iota)$, we refer to the part of the LTS that is reachable from $P(\iota)$ as the *state space*. Note that in defining the transition system in Definition 3.1.6, in the interpretation of an LPE a syntactic substitution is applied to the update expressions to define the reached state. This means that different closed syntactic expressions that correspond to the same semantic object, *e.g.*, $1+1$ and 2 for our assumed sort

Nat, result in different states. As stated by the lemma below, such states are always bisimilar.

Lemma 3.1.7. For all closed expressions $e, e' : D$ such that $[\![e \approx e']\!] = \mathbf{true}$ we have $P(e) \leftrightarrows P(e')$.

For any given state space we can therefore consider a *representative* state space where for each state a unique closed expression is chosen that is data equivalent. In examples we always consider the representative state space.

Example 3.1.8. Consider the following LPE, modelling a machine that alternates between two modes. The event that signals a switch between these two modes is modelled by action toggle; switching between modes happens after a number of clock cycles and is dependent on the mode in which the machine is running (3 cycles for one mode, 1 cycle for the other). The machine keeps track of its mode using a Boolean parameter s, and parameter n keeps track of the number of cycles left before switching modes.

$$\begin{aligned}\mathrm{Machine}(n : Nat, s : Bool) = {} & (n > 0) \to \mathsf{count} \,.\, \mathrm{Machine}(n-1, s) \\ & + (n \approx 0) \to \mathsf{toggle} \,.\, \mathrm{Machine}(\mathsf{if}(\neg s, 3, 1), \neg s)\end{aligned}$$

Note that the expression $\mathsf{if}(\neg s, 3, 1)$ models the reset of the clock cycle count upon switching modes. A representative state space of $\mathrm{Machine}(0, \mathsf{false})$, where the machine is initially off, is shown in Figure 3.1 (see page 58). □

3.1.3 A Process Algebra of Communicating Linear Process Equations

We define a minimal language to express parallelism and interaction of LPEs; the operators are taken from mCRL2 [63] and similar-styled process algebras. Let Comm be the set of *communication* expressions of the form $a_0 | \ldots | a_n \to c$ where $a_0, \ldots, a_n, c \in \Lambda$ are action labels.

Definition 3.1.9. The process algebra is defined as follows:

$$S ::= \Gamma_C(S) \mid \nabla_A(S) \mid \tau_H(S) \mid S \parallel S \mid P(\iota)$$

Here, $A \subseteq 2^{\Lambda \to \mathbb{N}}$ is a non-empty finite set of finite multi-sets of action labels, $H \subseteq \Lambda$ is a non-empty finite set of action labels and $C \subseteq \mathsf{Comm}$ is a finite set of *communications*. Finally, we have $P(\iota) \in \mathsf{P}$.

The set S contains all expressions of the process algebra. Operator Γ_C describes *communication*, ∇_A *action allowing*, τ_H *action hiding* and $\|$ *parallel composition*; the elementary objects are the processes, defined as LPEs.

The operational semantics of expressions in S are defined in Definition 3.1.12. We introduce three auxiliary functions on Ω that are used in the semantics. First of all, the auxiliary function γ_C defined below specifies the result of applying a set C of communications to a multi-action. A communication $a_0|\ldots|a_n \to c$ specifies that in a multi-action the actions $a_0(\mathbf{d}),\ldots,a_n(\mathbf{d})$ *synchronise* when carrying the same values, and result in action $c(\mathbf{d})$. Since multiple communications may be applicable at the same time, definition γ is naturally recursive.

Definition 3.1.10. Given $\omega \in \Omega$ we define γ_C, where $C \subseteq \mathsf{Comm}$, as follows:

$$\gamma_\emptyset(\omega) = \omega$$
$$\gamma_C(\omega) = \gamma_{C\setminus C_1}(\gamma_{C_1}(\omega)) \text{ for } C_1 \subset C$$
$$\gamma_{\{a_0|\ldots|a_n\to c\}}(\omega) = \begin{cases} \wr c(\mathbf{d})\wr + \gamma_{\{a_0|\ldots|a_n\to c\}}(\omega - \wr a_0(\mathbf{d}),\ldots,a_n(\mathbf{d})\wr) \\ \quad \text{if } \wr a_0(\mathbf{d}),\ldots,a_n(\mathbf{d})\wr \subseteq \omega \\ \omega \quad \text{otherwise} \end{cases}$$

For γ_C to be well-defined we require that the left-hand sides of the communications do not share labels. Furthermore, the action label on the right-hand side must not occur in any *other* left-hand side. For example $\gamma_{\{a|b\to c\}}(a|d|b) = c|d$, but $\gamma_{\{a|b\to c,a|d\to c\}}(a|d|b)$ and $\gamma_{\{a|b\to c,c\to d\}}(a|d|b)$ are not allowed.

Next, we define an auxiliary function $\theta_H(\omega)$ that defines the transformation that takes place when using the hiding operator $\tau_H(S)$. This function yields a multi-action in which all the action labels that occur in H are removed. Note that the multi-action becomes τ when all action labels are hidden.

Definition 3.1.11. Let $\omega \in \Omega$ and $H \subseteq \Lambda$. We define $\theta_H(\omega)$ as the multi-action $\omega' \in \Omega$ defined as:

$$\omega'(a(\mathbf{d})) = \begin{cases} 0 & \text{if } a \in H \\ \omega(a(\mathbf{d})) & \text{otherwise} \end{cases}$$

Finally, given a multi-action α we write $\underline{\alpha}$ to denote the multi-set of action labels that occur in α, *e.g.*, $\underline{a(3)|a(2)|b(5)} = \wr a:2, b\wr$. Formally, $\underline{a(e)} = \wr a\wr$, $\underline{\tau} = \wr\wr$ and $\underline{\alpha|\beta} = \underline{\alpha} + \underline{\beta}$. We define $\underline{\omega}$ for $\omega \in \Omega$ in a similar way.

Definition 3.1.12. The LTS $(\mathsf{S}, \Omega, \to)$ associated with expressions of S is defined by the rules below and the transition relation given in Definition 3.1.6 for each expression in P. For any $\omega, \omega' \in \Omega$, expressions P, P', Q, Q' of S and sets $C \subseteq \mathsf{Comm}$, $A \subseteq 2^{\Lambda\to\mathbb{N}}$ and $H \subseteq \Lambda$:

$$\textsc{Com}\ \frac{P \xrightarrow{\omega} P'}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega)} \Gamma_C(P')} \qquad \textsc{Allow}\ \frac{P \xrightarrow{\omega} P'}{\nabla_A(P) \xrightarrow{\omega} \nabla_A(P')}\ \underline{\omega} \in A \cup \{\tau\}$$

$$\textsc{Hide}\ \frac{P \xrightarrow{\omega} P'}{\tau_H(P) \xrightarrow{\theta_H(\omega)} \tau_H(P')} \qquad \textsc{Par}\ \frac{P \xrightarrow{\omega} P' \quad Q \xrightarrow{\omega'} Q'}{P \parallel Q \xrightarrow{\omega+\omega'} P' \parallel Q'}$$

$$\textsc{ParR}\ \frac{Q \xrightarrow{\omega} Q'}{P \parallel Q \xrightarrow{\omega} P \parallel Q'} \qquad \textsc{ParL}\ \frac{P \xrightarrow{\omega} P'}{P \parallel Q \xrightarrow{\omega} P' \parallel Q}$$

Note that for ALLOW the condition $\underline{\omega} \in A \cup \{\tau\}$ must be satisfied in order for the rule to be applicable.

Observe that PARL and PARR are distinct rules from PAR. Rule PAR expresses that a transition from both P and Q happen simultaneously, whereas rules PARR and PARL cover the case in which only one of the two processes involved in the parallel composition executes a transition. Furthermore, observe that it is not possible to disallow the occurrence of a τ-transition, because the condition $\underline{\tau} \in A \cup \{\tau\}$ rule ALLOW is true for any A since $\underline{\tau} = \tau$.

Example 3.1.13. Consider the following LPE that models a drill component in which each toggle action leads to a drill action.

$$\begin{aligned} \mathrm{Drill}(t : \mathit{Bool}) = (\neg t) &\to \mathsf{toggle} \,.\, \mathrm{Drill}(\mathsf{true}) \\ + \quad (t) &\to \mathsf{drill} \,.\, \mathrm{Drill}(\mathsf{false}) \end{aligned}$$

Suppose that we wish to study the interaction of LPEs Machine of Example 3.1.8 and Drill, assuming that their toggle actions must synchronise, resulting in a toggle' action. Let $C = \{\mathsf{toggle}|\mathsf{toggle} \to \mathsf{toggle}'\}$ be the communication that specifies this synchronisation, and let $A = \{\tau\mathsf{toggle}', \tau\mathsf{drill}, \tau\mathsf{count}\}$ be the set of multi-action labels we allow. The interaction between LPEs Machine and Drill can be specified by the expression $\nabla_A(\Gamma_C(\mathrm{Machine}(0,\mathsf{false}) \parallel \mathrm{Drill}(\mathsf{false})))$ in the algebra. An example derivation is depicted below, invoking (from top to bottom) rule PAR, then COM and finally ALLOW:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{Machine}(0,\mathsf{false}) \xrightarrow{\tau\mathsf{toggle}} \mathrm{Machine}(3,\mathsf{true}) \qquad \mathrm{Drill}(\mathsf{false}) \xrightarrow{\tau\mathsf{toggle}} \mathrm{Drill}(\mathsf{true})}{\mathrm{Machine}(0,\mathsf{false}) \parallel \mathrm{Drill}(\mathsf{false}) \xrightarrow{\tau\mathsf{toggle:2}} \mathrm{Machine}(3,\mathsf{true}) \parallel \mathrm{Drill}(\mathsf{true})}}{\Gamma_C(\mathrm{Machine}(0,\mathsf{false}) \parallel \mathrm{Drill}(\mathsf{false})) \xrightarrow{\tau\mathsf{toggle}'} \Gamma_C(\mathrm{Machine}(3,\mathsf{true}) \parallel \mathrm{Drill}(\mathsf{true}))}}{\nabla_A(\Gamma_C(\mathrm{Machine}(0,\mathsf{false}) \parallel \mathrm{Drill}(\mathsf{false}))) \xrightarrow{\tau\mathsf{toggle}'} \nabla_A(\Gamma_C(\mathrm{Machine}(3,\mathsf{true}) \parallel \mathrm{Drill}(\mathsf{true})))}$$

Observe that in the last step in the derivation, rule ALLOW is applicable since $\underline{\wr \mathsf{toggle}' \wr} \in A \cup \{\wr, \wr\}$. Note that we cannot derive a $\wr \mathsf{toggle} \wr$ transition for the expression $\nabla_A(\Gamma_C(\text{Machine}(0, \mathsf{false}) \parallel \text{Drill}(\mathsf{false})))$, even though by, *e.g.*, PARL, we can derive $\text{Machine}(0, \mathsf{false}) \parallel \text{Drill}(\mathsf{false}) \xrightarrow{\wr \mathsf{toggle} \wr} \text{Machine}(3, \mathsf{true}) \parallel \text{Drill}(\mathsf{false})$. The reason for this is that $\underline{\wr \mathsf{toggle} \wr} \notin A \cup \{\wr, \wr\}$. □

3.2 The Decomposition Problem

The state space of a monolithical LPE may grow quite large and generating that state space may either take too long or require too much memory. We are therefore interested in decomposing an LPE into two or more LPEs, where the latter are referred to as *components*, such that the state spaces of the resulting components are smaller than that of the original state space. Such a decomposition is considered *valid* iff the original state space is strongly bisimilar to the state space of these components when combined under a suitable context (*i.e.*, an expression with a 'hole') that specifies how to combine the components. We formalise this problem as follows.

Definition 3.2.1. Let $P(\vec{d} : \vec{D}) = \phi$ be an LPE and $\vec{t} : \vec{D}$ a closed expression. The LPEs $P_0(\vec{d}_{|I_0} : \vec{D}_{|I_0}) = \phi_0$ to $P_n(\vec{d}_{|I_n} : \vec{D}_{|I_n}) = \phi_n$, for sets of indices $I_0, \ldots, I_n \subseteq \mathbb{N}$, are a valid *decomposition* of P and $\vec{t}$ iff there is a context C such that:

$$P(\vec{t}) \leftrightarrows \mathsf{C}[P_0(\vec{t}_{|I_0}) \parallel \ldots \parallel P_n(\vec{t}_{|I_n})]$$

where $\mathsf{C}[P_0(\vec{t}_{|I_0}) \parallel \ldots \parallel P_n(\vec{t}_{|I_n})]$ is an expression in S. We refer to the expression $\mathsf{C}[P_0(\vec{t}_{|I_0}) \parallel \ldots \parallel P_n(\vec{t}_{|I_n})]$ as the *composition*.

In the next sections, we will show that a suitable context C can be constructed using the operators from S, and we define a decomposition technique that results in exactly two components (a *cleave*). The technique can, in principle, be applied recursively to these two components. The primary benefit of a valid decomposition is that a state space that is equivalent to the original state space can be obtained using *compositional minimisation*, which can result in a state space that is immediately be significantly smaller than the state space resulting from monolithic exploration. First, the state space of each component is derived separately. The composition can then be derived from the component state spaces, exploiting the rules of the operational semantics. The component state spaces can be minimised modulo bisimilarity, which is a congruence with respect to the operators of S before deriving the results of the composition expression.

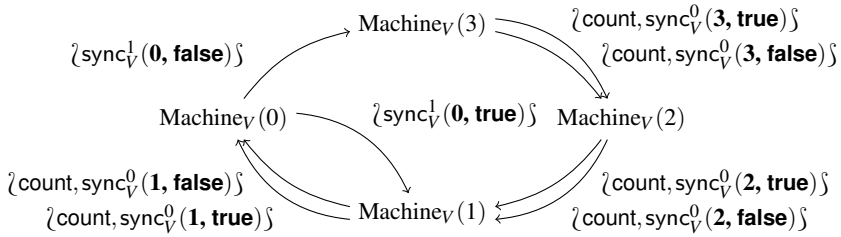
3.3 A Solution to the Decomposition Problem

A basic observation that we exploit in our solution to the decomposition problem is that when hiding label c in a multi-action $\alpha|c$, we are left with multi-action α, provided that c does not occur in α. When the multi-action α is an event that is possible in a monolithic LPE and the label c is the result of a communication between two components, we can effectively exchange information between multiple components, without this information becoming visible externally. The example below illustrates the idea using a naive but valid solution to the decomposition technique on the LPE of Example 3.1.8.

Example 3.3.1. Reconsider the LPE *Machine* we defined earlier, and consider the two components depicted below.

$$\begin{aligned}
\mathrm{Machine}_V(n:Nat) = &\sum_{s:Bool} (n>0) \to \mathsf{count}|\mathsf{sync}^0_V(n,s)\,.\,\mathrm{Machine}_V(n-1)\\
&+ \sum_{s:Bool} (n \approx 0) \to \mathsf{sync}^1_V(n,s)\,.\,\mathrm{Machine}_V(\mathsf{if}(\neg s,3,1))\\
\mathrm{Machine}_W(s:Bool) = &\sum_{n:Nat} (n>0) \to \mathsf{sync}^0_W(n,s)\,.\,\mathrm{Machine}_W(s)\\
&+ \sum_{n:Nat} (n \approx 0) \to \mathsf{toggle}|\mathsf{sync}^1_W(n,s)\,.\,\mathrm{Machine}_W(\neg s)
\end{aligned}$$

Each component describes part of the behaviour and knows the value of parameter n or s, but not the other. To cater for this, it is 'over-approximated' by a sum variable. The state space of $\mathrm{Machine}_V(0)$ is shown below. The synchronisation actions sync expose the non-deterministically chosen values of the unknown parameters.



Enforcing synchronisation of the sync actions, the context C can be chosen as follows to achieve a valid decomposition:

$$\begin{aligned}
&\nabla_{\{⦇\mathsf{toggle}⦈,⦇\mathsf{count}⦈\}}(\tau_{\{\mathsf{sync}^0,\mathsf{sync}^1\}}\\
&\quad(\Gamma_{\{\mathsf{sync}^0_V|\mathsf{sync}^0_W\to\mathsf{sync}^0,\mathsf{sync}^1_V|\mathsf{sync}^1_W\to\mathsf{sync}^1\}}\\
&\qquad(\mathrm{Machine}_V(0) \parallel \mathrm{Machine}_W(\mathsf{false}))))
\end{aligned}$$

Unfortunately the state space of $\text{Machine}_W(\mathsf{false})$ in the above example is infinitely branching and it has no finite state space that is strongly bisimilar to it, rendering the decomposition useless in practice. We will subsequently develop a more robust solution.

3.3.1 Separation Tuples

To obtain a useful decomposition it can be beneficial to reduce the number of parameters that occur in the synchronisation actions, because these then become a visible part of the transitions in the state spaces of the individual components. In the worst case, as illustrated by LPE Machine_W of Example 3.3.1, synchronisation actions lead to a component having an infinite state space despite the fact that the state space of the original LPE is finite.

One observation we exploit is that in some cases we can actually remove the synchronisation for summands completely. For instance, in the first summand of Machine in Example 3.1.8, the value of parameter s remains unchanged and the condition is only an expression containing parameter n. We refer to summands with such a property as *independent* summands, whereas the other summands are dependent. When defining the context C, we can allow a component to execute multi-actions of its independent summands without enforcing a synchronisation with the other component. This allows, for instance, component Machine_V to independently execute (multi-)action count without synchronising the values of s and n with Machine_W.

We must ensure that each dependent summand of the monolithic LPE is covered by both components that we extract from the LPE. However, an independent summand of one component does not need a corresponding summand in the other component. Therefore, the summands that we extract for a given component are identified by a set of indices J of the summands of the monolithic LPE. Of these, we furthermore can identify summands that are dependent and summands that are independent. The indices for the latter are collected in a set K.

A third observation that can be utilised is that for the dependent summands, there is some degree of flexibility for deciding which component will contribute to which part of the summand of the monolithic LPE. More specifically, by carefully distributing the enabling condition c and action expression α of a summand of the monolithic LPE over the two components, the amount of information that needs to be exchanged between these two components when they execute their respective summands, can be minimised. That is, the synchronisation of 'missing' parameters, *i.e.*, the parameters of the other component, might be avoided when the condition and action expressions of one component no longer contain that parameter.

The final observation is that the synchronisation actions of dependent summands can be used to synchronise the result of arbitrary expressions instead of only process

parameters. This can be used when distributing equality conditions over the two components, *i.e.*, conditions of the form $e \approx e'$, where one side of the equality (*e.g.*, expression e) can be put into the synchronisation action of one component and the other side of the equality (expression e') into the other component. We use the term *synchronisation expression* to refer to the expression passed along an argument to the synchronisation action.

Note that the way we distribute the list of process parameters of the monolithic LPE over the two components may affect which summands can be considered independent. For instance, had we decided to assign the (multi-)action count to Machine_W and toggle to Machine_V, we would not be able to declare count's summand independent. Consequently, the set of process parameter indices U, assigned to a component, and the set K are mutually dependent.

To capture this relation, we introduce the concept of a *separation tuple*. The concept of a separation tuple, a 6-tuple which we introduce below, formalises the required relation between the sets of indices for independent summands K, summands J and process parameters U, and the conditions c, and action α and a vector of data expressions $\vec{h}$ that we call synchronisation expressions of a component. We use indexed sets to define the condition, action and update expressions for every dependent summand. For elements in an indexed set we use subscript notation to indicate the index of that element.

Definition 3.3.2. Let $P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE. A *separation tuple* for P is a 6-tuple $(U, K, J, c^U, \alpha^U, \vec{h}^U)$ where $U \subseteq \mathbb{N}$ is a set of parameter indices, $K \subseteq J \subseteq I$ are two sets of summand indices, and c^U, α^U and $\vec{h}^U$ are sets of condition, action and synchronisation expressions respectively, indexed by indices from $J \setminus K$. We require that for all $i \in (J \setminus K)$ it holds that $\mathsf{FV}(c_i^U) \cup \mathsf{FV}(\alpha_i^U) \cup \mathsf{FV}(\vec{h}_i^U) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$, and for all $i \in K$ it holds that $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_{i|U}) \subseteq \mathsf{Vars}(\vec{d}_{|U}) \cup \{e_i\}$.

A separation tuple induces an LPE, where $U^c = \mathbb{N} \setminus U$, as follows:

$$P_U(\vec{d}_{|U} : \vec{D}_{|U}) = \sum_{i \in (J \setminus K)} \sum_{e_i : E_i, \vec{d}_{|U^c} : \vec{D}_{|U^c}} c_i^U \to \alpha_i^U | \mathsf{sync}_U^i(\vec{h}_i^U) \,.\, P_U(\vec{g}_{i|U}) + \sum_{i \in K} \sum_{e_i : E_i} c_i \to \alpha_i | \mathsf{tag} \,.\, P_U(\vec{g}_{i|U})$$

We assume that action label sync_U^i, for any $i \in I$, and label tag does not occur in α_j, for any $j \in I$, to ensure that these action labels are fresh.

Observe that for independent summands the action label is extended with a tag action in Definition 3.3.2. This label is (only) needed to properly deal with overlapping multi-actions, as we illustrate below in Example 3.3.3.

Example 3.3.3. Consider the following LPE.

$$\begin{aligned}\mathrm{P}(x:Bool,y:Bool) = {} & x \to a\,.\,\mathrm{P}(\mathsf{false},y)\\ & + y \to b\,.\,\mathrm{P}(x,\mathsf{false})\\ & + (x \wedge \neg y) \to a|b\,.\,\mathrm{P}(\mathsf{false},\mathsf{false})\end{aligned}$$

Suppose we decompose LPE P with indices $I = \{0,1,2\}$ using the separation tuples $(V,\{0\},\{0,2\},\{x_2\},\{a_2\},\{\langle\rangle_2\})$ and $(W,\{1\},\{1,2\},\{(\neg y)_2\},\{b_2\},\{\langle\rangle_2\})$, where $V = \{0\}$ and $W = \{1\}$, and assuming that the summands of P are indexed from top to bottom by 0, 1 and 2 respectively. Now assume that we had omitted the tag action in Definition 3.3.2, in which case these separation tuples would induce the following LPEs:

$$\begin{aligned}P_V(x:Bool) = {} & x \to a\,.\,P_V(\mathsf{false})\\ & + x \to a|\mathsf{sync}^2_V\,.\,P_V(\mathsf{false})\\ P_W(y:Bool) = {} & y \to b\,.\,P_W(\mathsf{false})\\ & + (\neg y) \to b|\mathsf{sync}^2_W\,.\,P_W(\mathsf{false})\end{aligned}$$

Observe that both $P_V(\mathsf{true}) \xrightarrow{\wr a \wr} P_V(\mathsf{false})$ and $P_W(\mathsf{true}) \xrightarrow{\wr b \wr} P_W(\mathsf{false})$ are transitions for these components. This also means that due to (among others) rule PAR, $P_V(\mathsf{true}) \parallel P_W(\mathsf{true})$ can perform action $\wr a,b \wr$. Note that $P(\mathsf{true},\mathsf{true})$ does *not* have an outgoing transition labelled with $\wr a,b \wr$, but (the reachable) process $P(\mathsf{true},\mathsf{false})$ *does* have an outgoing $\wr a,b \wr$ transition. There is, however, no composition expression that prevents $\wr a,b \wr$ in $P_V(\mathsf{true}) \parallel P_W(\mathsf{true})$ and allows $\wr a,b \wr$ in $P_V(\mathsf{true}) \parallel P_W(\mathsf{false})$. The tag label provides the tools for making this distinction by only allowing at most one tag action to be present, and therefore disallowing $\wr a,b,\mathsf{tag},\mathsf{tag} \wr$. □

The components, induced by two separation tuples, can be (re)combined in a context that enforces synchronisation of the sync events and which hides their communication trace. This ensures that all actions left can be traced back to the monolithic LPE from which the components are derived. Under specific conditions, this is achieved by the following context.

Definition 3.3.4. Let $P(\vec{d}:\vec{D}) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i\,.\,P(\vec{g}_i)$ be an LPE with the separation tuples $(V,K^V,J^V,c^V,\alpha^V,\vec{h}^V)$ and $(W,K^W,J^W,c^W,\alpha^W,\vec{h}^W)$ for P. Let $P_V(\vec{d}_{|V}:\vec{D}_{|V}) = \phi_V$ and $P_W(\vec{d}_{|W}:\vec{D}_{|W}) = \phi_W$ be the induced LPEs according to Definition 3.3.2. Let $\vec{t}:\vec{D}$ be a closed expression. Then the composition expression is

defined as:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \mid i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \mid i\in(K^V\cup K^W)\}}(\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\\ \Gamma_{\{\mathsf{sync}^i_V|\mathsf{sync}^i_W\to\mathsf{sync}^i \mid i\in I\}}(P_V(\vec{t}_{|V}) \parallel P_W(\vec{t}_{|W})))))$$

Before we proceed to identify the conditions under which two separation tuples induce a valid decomposition using the above context, we revisit Example 3.1.8 to illustrate the concepts introduced so far.

Example 3.3.5. Reconsider the LPE presented in Example 3.1.8 with $V=\{0\}$ and $W=\{1\}$. The separation tuple $(V,\{0\},\{0,1\},\{(n\approx 0)_1\},\{\tau_1\},\{\langle s\rangle_1\})$ and the tuple $(W,\emptyset,\{1\},\{\mathsf{true}_1\},\{\mathsf{toggle}_1\},\{\langle s\rangle_1\})$ for Machine induce component $\mathrm{Machine}_V$ and $\mathrm{Machine}_W$ respectively.

$$\begin{aligned}\mathrm{Machine}_V(n:Nat) = \quad & (n>0)\to \mathsf{count}|\mathsf{tag}\,.\,\mathrm{Machine}_V(n-1)\\ &+\sum_{s:Bool}(n\approx 0)\to\tau|\mathsf{sync}^1_V(s)\,.\,\mathrm{Machine}_V(\mathsf{if}(\neg s,3,1))\\ \mathrm{Machine}_W(s:Bool) = \quad & \mathsf{true}\to\mathsf{toggle}|\mathsf{sync}^1_W(s)\,.\,\mathrm{Machine}_W(\neg s)\end{aligned}$$

Note that we omitted the $\sum$-operator in the first summand of $\mathrm{Machine}_V$ since sum variable s does not occur as a free variable in the expressions; for similar reasons, the $\sum$-operator is omitted in $\mathrm{Machine}_W$. The state spaces of components $\mathrm{Machine}_V(0)$ and $\mathrm{Machine}_W(\mathsf{false})$ are shown in Figure 3.2. We obtain the following composition according to Definition 3.3.4:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\wr\mathsf{toggle}\wr,\wr\mathsf{count}\wr,\wr\mathsf{count},\mathsf{tag}\wr\}}(\tau_{\{\mathsf{sync}^0,\mathsf{sync}^1\}}(\\ \Gamma_{\{\mathsf{sync}^0_V|\mathsf{sync}^0_W\to\mathsf{sync}^0,\mathsf{sync}^1_V|\mathsf{sync}^1_W\to\mathsf{sync}^1\}}(\mathrm{Machine}_V(0) \parallel \mathrm{Machine}_W(\mathsf{false})))))$$

This composition expression is strongly bisimilar to $\mathrm{Machine}(0,\mathsf{false})$ shown in Figure 3.1. Note that the state space of $\mathrm{Machine}_V(0)$ has four states and transitions, and the state space of $\mathrm{Machine}_W(\mathsf{false})$ has two states and transitions, which are both smaller than the original state space. Their composition has the same size as the original state space and no further minimisation can be achieved (note that the state space of Figure 3.1 is already minimal). □

3.3.2 Cleave Correctness Criteria

It may be clear that not every decomposition which satisfies Definition 3.3.4 yields a *valid* decomposition (in the sense of Definition 3.2.1). For example, replacing the condition expression true in Example 3.3.5 of the summand in P_W by false would not

result in a valid decomposition. Our aim in this section is to present the necessary and sufficient conditions to establish that the state space of the monolithic LPE is bisimilar to the state space of the composition expression resulting from Definition 3.3.4. We prove this in Theorem 3.3.12.

Consider a decomposition of an LPE P according to Definition 3.3.4, induced by separation tuples $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$. We abbreviate the composition expression of Definition 3.3.4 by $\mathsf{C}[P_V(\vec{d}_{|V}) || P_W(\vec{d}_{|W})]$. Recall that components P_V and P_W yield a valid decomposition of P if there is a bisimulation relation between $P(\vec{d})$ and $\mathsf{C}[P_V(\vec{d}_{|V}) || P_W(\vec{d}_{|W})]$. A bisimulation relation requires that related states can mimic each other's steps. Since three LPEs are involved (the LPE P and the two interacting components, induced by the separation tuples), we must consider situations that can emerge from any of these three LPEs executing a (multi-)action for states related by the bisimulation relation.

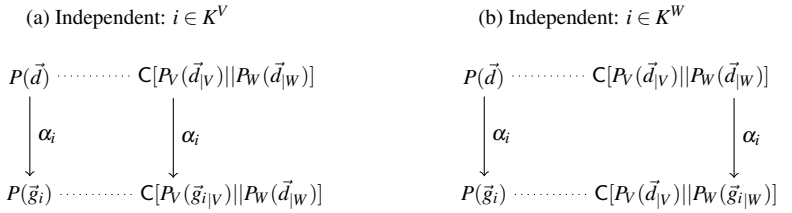


Figure 3.3: Two of the possible situations that must be considered when showing the validity of the decomposition of Definition 3.3.4: the execution of independent summands depicted in situations (a) and (b).

Two of the three relevant scenarios that must be considered are depicted in Figure 3.3. Note that in all relevant scenarios, the initiative of the transition may be with either $P(\vec{d})$, or with the composition $\mathsf{C}[P_V(\vec{d}_{|V}) || P_W(\vec{d}_{|W})]$.

Suppose that the monolithic LPE P can take a step due to some summand $i \in I$, for which also $i \in K^V$. In that case—case (a) in Figure 3.3—Definition 3.3.4 guarantees that the free variables of their condition, action and update expressions are taken from $\vec{d}_{|V}$; (multi-)action α_i matches (multi-)action $\alpha_i|\mathsf{tag}$ after hiding tag. However, this is not sufficient to guarantee full independence of both components: what may happen is that the execution of a summand that is assumed to be independent still modifies the value of a process parameter of the other component, violating the idea of independence, and resulting in a target state in the composition that cannot be related to the target state of the monolithic LPE. In order to guarantee true independence, we must require that the W-projection on the update expression $\vec{g}_i$ of P does not modify the corresponding parameters. Case (b) in Figure 3.3 is dual. Formally, we require

(IND): for all $r \in K^V$ we have $\vec{g}_{r|W} = \vec{d}_{|W}$ and for all $r \in K^W$ we demand $\vec{g}_{r|V} = \vec{d}_{|V}$. Note that in case K^V and K^W overlap, condition (IND) guarantees that the involved summands only induce self-loops since whenever $\vec{g}_{r|W} = \vec{d}_{|W}$ and $\vec{g}_{r|V} = \vec{d}_{|W}$ then $\vec{g}_r = \vec{d}$; thus no updates take place. Finally, observe that (IND) is also a sufficient condition for the monolithic LPE P to match a (multi-)action α_r in both cases (a) and (b) of Figure 3.3, because the matching summand must occur in either P_V or P_W.

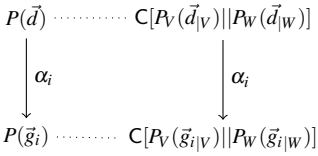


Figure 3.4: The third possible situation that must be considered when showing the validity of the decomposition of Definition 3.3.4: the synchronous execution of summands.

The more complex scenario that must be considered is when P_V and P_W (must) synchronise to mimic the behaviour of P; see Figure 3.4 due to the structure of C. Suppose again that the monolithic LPE P can execute an α_i action due to summand $i \in I$, but in this case, neither $i \in K^V$, nor $i \in K^W$. First, observe that the only option to match the behaviour of this summand is if a component covers at least all summands not already covered by the other component. We must therefore require at least the following condition (SYN): $J^V = I \setminus K^W$ and $J^W = I \setminus K^V$. It then follows from $K^V \subseteq J^V$ (and $K^W \subseteq J^W$) that $J^V \cap J^W$ are the summands that induce synchronisation and that these are disjoint from the independent summands; which are in $K^V \cup K^W$.

Second, observe that the enabledness of summand i in P depends on the enabling condition c_i. Consequently, if c_i holds true, then the i-indexed conditions c_i^V and c_i^W must also hold true. Moreover, since we are dealing with dependent summands, the multi-action expression $\alpha_i^V | \alpha_i^W$ must reduce to α_i under these conditions. Also the additional synchronisation vectors $\vec{h}^V$ and $\vec{h}^W$ must agree, for otherwise the sync actions of both components cannot participate in the synchronisation. Note that we do not need to explicitly require relating the update expressions of P and the components P_V and P_W resulting from the execution of their i-indexed summands, since this property is already guaranteed by construction; see Definition 3.3.2. We collectively refer to the above requirements by condition (ORI).

Vice versa, whenever both components can simultaneously execute their i-indexed summand, we must ensure that also the monolithic LPE P can execute its i-indexed summand. Condition (COM) ensures that this requirement is met. Note that P_V and

P_W only synchronise on summands with equal indices due to the synchronisation on sync actions that is enforced. A technical complication in formalising requirement (COM), however, is that the sum variables of the individual components carry the same name in all three LPEs. In particular, from the fact that both individual components can successfully synchronise, we cannot deduce a unique value assigned to these homonymous sum-variables. We must therefore also ensure that the update expressions of the components, resulting from executing the r-indexed summands, indeed is the same as could have resulted from executing the r-indexed summand in P. Since this property is *not* guaranteed by the construction of Definition 3.3.2, there is a need to explicitly require it to hold.

A pair of separation tuples of P satisfying the above requirements is called a *cleave* of P. Below, we formalise this notion, together with the requirements we informally introduced above. We defer a formal proof of correctness of these requirements to the end of this section.

Definition 3.3.6. Let $P(\vec{d}:\vec{D}) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE with the separation tuples $(V,K^V,J^V,c^V,\alpha^V,\vec{h}^V)$ and $(W,K^W,J^W,c^W,\alpha^W,\vec{h}^W)$ for P as defined in Definition 3.3.2. The two separation tuples are a *cleave* of P iff the following requirements hold.

SYN. $J^V = I \setminus K^W$ and $J^W = I \setminus K^V$.

IND. For all $r \in K^V$, $\vec{g}_{r|W} = \vec{d}_{|W}$, and for all $r \in K^W$, $\vec{g}_{r|V} = \vec{d}_{|V}$.

ORI. For all $r \in (J^V \cap J^W)$ and substitutions σ satisfying $[\![\sigma(c_r)]\!]$, also:

- $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma(c_r^W)]\!]$, and
- $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma(\vec{h}_r^W)]\!]$, and
- $[\![\sigma(\alpha_r^V|\alpha_r^W)]\!] = [\![\sigma(\alpha_r)]\!]$.

COM. For all $r \in (J^V \cap J^W)$ and substitutions σ and σ' satisfying $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma'(c_r^W)]\!]$ and $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$, there is a substitution ρ such that $[\![\rho(\vec{d}_{|V})]\!] = [\![\sigma(\vec{d}_{|V})]\!]$ and $[\![\rho(\vec{d}_{|W})]\!] = [\![\sigma'(\vec{d}_{|W})]\!]$ and:

- $[\![\rho(c_r)]\!]$, and
- $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$, and
- $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and
- $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$.

Example 3.3.7. We argue that the separation tuples inducing the decomposition obtained in Example 3.3.5 are a cleave indeed. First of all, the requirements SYN and IND can be checked quite easily. The requirements ORI and COM both have

to be checked for the summand with index one. Consider the requirement ORI with a substitution σ assigning any value to n (and any value to other variables due to totality) such that $[\![\sigma(n \approx 0)]\!]$ holds. It follows directly that both $[\![\sigma(n \approx 0)]\!]$ and $[\![\sigma(\mathsf{true})]\!]$ hold. Furthermore, $\langle s \rangle$ is the same synchronisation expression on both sides and $[\![\sigma(\tau|\mathsf{toggle})]\!] = [\![\sigma(\mathsf{toggle})]\!]$ by definition. For the requirement COM consider any two substitutions σ and σ' such that both $[\![\sigma(n \approx 0)]\!]$ and $[\![\sigma'(\mathsf{true})]\!]$ hold and $[\![\sigma(\langle s \rangle)]\!] = [\![\sigma'(\langle s \rangle)]\!]$. For substitution ρ we can choose n to be zero and s to be equal to $\sigma(s)$. The most interesting observation is that then indeed $[\![\sigma(\mathsf{if}(\neg s, 3, 1))]\!] = [\![\rho(\mathsf{if}(\neg s, 3, 1))]\!]$ and that $[\![\sigma'(\neg s)]\!] = [\![\rho(\neg s)]\!]$. The other conditions are also satisfied and thus this is a cleave. We can also observe that leaving out the synchronisation of s does not yield a cleave since there is no substitution ρ meeting the conditions in COM when substitutions σ and σ' disagree on the value of s. □

Informally, we have already argued that the decomposition yields a state space that is bisimilar to the original monolithic LPE. We finish this section with a formal claim stating that a cleave induces a valid decomposition of a monolithic LPE. For this, we first introduce several auxiliary results and concepts. In particular, the proof of correctness of our claim relies on the notion of bisimulation *up to* [105].

Definition 3.3.8. Let $\mathscr{L} = (S, Act, \rightarrow)$ be an LTS. A binary relation $R \subseteq S \times S$ is a *strong bisimulation up to* $\leftrightarrows$ iff for all $s\,R\,t$ it holds that:

- if $s \xrightarrow{\omega} s'$ then there is a state $t' \in S$ such that $t \xrightarrow{\omega} t'$ and $s' \leftrightarrows R \leftrightarrows t'$.
- if $t \xrightarrow{\omega} t'$ then there is a state $s' \in S$ such that $s \xrightarrow{\omega} s'$ and $t' \leftrightarrows R \leftrightarrows s'$.

where the notation $\leftrightarrows R \leftrightarrows$ denotes the *relational composition*, which is defined as $\leftrightarrows R \leftrightarrows \; = \{(s,t) \in S \times S \mid \exists s', t' \in S : s \leftrightarrows s' \wedge s' \,R\, t' \wedge t' \leftrightarrows t\}$.

Proposition 3.3.9 (See [105])**.** If R is a strong bisimulation up to $\leftrightarrows$ then $R \subseteq \leftrightarrows$

This result establishes that if R is a strong bisimulation up to $\leftrightarrows$ then for any pair $(s,t) \in R$ we can conclude that $s \leftrightarrows t$.

We introduce two technical auxiliary lemmas to relate the transition induced by some expression $P \in \mathsf{S}$ to the transitions induced by applying the allow, hide and communication operators, in the same order as the composition expression defined in Definition 3.3.4, to P. The reader may skip their proofs because these are standard, but we nevertheless include these for the sake of completeness.

Lemma 3.3.10. Given expressions $P, Q \in \mathsf{S}$, a set of multi-sets of action labels $A \subseteq 2^{\Lambda \to \mathbb{N}}$, two sets of action labels $H', H \subseteq \Lambda$, a set of communications $C \subseteq \mathsf{Comm}$. If $P \xrightarrow{\omega'} Q$ and $\underline{\theta_H(\gamma_C(\omega'))} \in A'$ with $A' = A \cup \{\langle\rangle\}$ then:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$$

Proof. We can derive the following:

$$\textsc{Hide}\ \frac{\textsc{Allow}\ \dfrac{\textsc{Hide}\ \dfrac{\textsc{Com}\ \dfrac{P \xrightarrow{\omega'} Q}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega')} \Gamma_C(Q)}}{\tau_H(\Gamma_C(P)) \xrightarrow{\theta_H(\gamma_C(\omega'))} \tau_H(\Gamma_C(Q))}}{\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\theta_H(\gamma_C(\omega'))} \nabla_A(\tau_H(\Gamma_C(Q)))}\ \underline{\theta_H(\gamma_C(\omega'))} \in A_{\wr\wr}}{\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))}$$

□

Lemma 3.3.11. Given expressions $P, Q' \in \mathsf{S}$, a set of multi-sets of action labels $A \subseteq 2^{\Lambda \to \mathbb{N}}$, two sets of action labels $H', H \subseteq \Lambda$ and a set of communications $C \subseteq \mathsf{Comm}$. If:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q'$$

then there are $Q \in \mathsf{S}$ and $\omega' \in \Omega$ such that $Q' = \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$, and $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega')))$, $P \xrightarrow{\omega'} Q$ and $\underline{\theta_H(\gamma_C(\omega'))} \in A \cup \{\wr\wr\}$.

Proof. Pick arbitrary expressions $P, Q' \in \mathsf{S}$, a set of multi-sets of action labels $A \subseteq 2^{\Lambda \to \mathbb{N}}$, two sets of action labels $H', H \subseteq \Lambda$ and a set of communications $C \subseteq \mathsf{Comm}$. Assume that $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q'$. By the syntactic structure of $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P))))$, we must conclude that transition $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q'$ must be due to rule HIDE and Q' must be of the shape $\tau_{H'}(Q_1)$ for some expression Q_1 and $\omega = \theta_{H'}(\omega_1)$ for some ω_1:

$$\textsc{Hide}\ \frac{\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\omega_1} Q_1}{\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\omega_1)} \tau_{H'}(Q_1)}$$

Now, we find that only the ALLOW rule has a conclusion that permits us to derive $\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega_1} Q_1$, in which case Q_1 must be of the shape $\nabla_A(Q_2)$ for some expression Q_2, and $\underline{\omega_1} \in A \cup \{\wr\wr\}$:

$$\textsc{Allow}\ \frac{\tau_H(\Gamma_C(P)) \xrightarrow{\omega_1} Q_2}{\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\omega_1} \nabla_A(Q_2)}\ \underline{\omega_1} \in A \cup \{\wr\wr\}$$

Observe that Q' is also of the shape $\tau_{H'}(\nabla_A(Q_2))$. Again, a transition $\tau_H(\Gamma_C(P)))) \xrightarrow{\omega_1} Q_2$ can only be derived when Q_2 is of the shape $\tau_H(Q_3)$, for some expression Q_3, and $\omega_1 = \theta_H(\omega_2)$ for some ω_2:

$$\textsc{Hide}\ \frac{\Gamma_C(P) \xrightarrow{\omega_2} Q_3}{\tau_H(\Gamma_C(P)) \xrightarrow{\theta_H(\omega_2)} \tau_H(Q_3)}$$

Note that then Q' is of the shape $\tau_{H'}(\nabla_A(\tau_H(Q_3)))$. Finally, only the COM rule allows us to derive $\Gamma_C(P) \xrightarrow{\omega_2} Q_3$, in which case Q_3 is of the form $\Gamma_C(Q_4)$, for some Q_4, and $\omega_2 = \gamma_C(\omega_3)$ for some ω_3.

$$\textsc{Com}\ \frac{P \xrightarrow{\omega_3} Q_4}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega_3)} \Gamma_C(Q_4)}$$

For this derivation to exist it must be true that for some expression Q_4, transition $P \xrightarrow{\omega_3} Q_4$ exists and that Q' is of the form $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q_4))))$, and $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega_3)))$ for some ω_3. Furthermore, it must hold that $\underline{\omega_1} = \underline{\theta_H(\gamma_C(\omega_3))} \in A \cup \{\lightning\}$. □

We now formalise the correctness of the cleave in Theorem 3.3.12. The proof of this Theorem is a formalisation of the informal reasoning presented in the beginning of this section.

Theorem 3.3.12. Let $P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE with the separation tuples $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ for P that are a cleave as defined in Definition 3.3.6. For every closed expression $\vec{\iota} : \vec{D}$ the composition expression defined in Definition 3.3.4 is strongly bisimilar to $P(\vec{\iota})$ and, hence, a valid decomposition according to Definition 3.2.1.

Proof. Let $A = \{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i | \mathsf{tag}} \mid i \in (K_V \cup K_W)\}$, $H' = \{\mathsf{tag}\}$, $H = \{\mathsf{sync}^i \mid i \in I\}$ and $C = \{\mathsf{sync}^i_V | \mathsf{sync}^i_W \to \mathsf{sync}^i \mid i \in I\}$. Let R be the least relation such that $P(\vec{\iota}') \, R \, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}'_{|V}) \parallel P_W(\vec{\iota}'_{|W})))))$ for closed expressions $\vec{\iota}' : \vec{D}$. We show that R is a strong bisimulation relation up to $\leftrightarrows$.

Pick any closed expression $\vec{\iota} : \vec{D}$ and assume that $P(\vec{\iota}) \; R \; \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})))))$.

- Assume that $P(\vec{\iota}) \xrightarrow{\omega} Q'$. Then there is an index $r \in I$ and substitution σ such that $\sigma(\vec{d}) = \vec{\iota}$ for which it holds that $[\![\sigma(c_r)]\!]$, $\omega = [\![\sigma(\alpha_r)]\!]$ and $Q' = P(\sigma(\vec{g}_r))$. There are three cases to consider based on the index r.
  - Case $r \in K_V$. This is essentially the case visualised in Figure 3.3 (a), where the initiative comes from the monolithic process. We derive the transition $P_V(\vec{\iota}_{|V}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{tag}]\!]} P_V(\sigma(\vec{g}_{r|V}))$, because $[\![\sigma(c_r)]\!]$ holds. Rule PARL allows us to derive the transition $P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{tag}]\!]} P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\vec{\iota}_{|W})$. By definition, $\theta_{H'}(\theta_H(\gamma_C(\sigma(\alpha_r)|\mathsf{tag}))) = \sigma(\alpha_r)$ and $\underline{\sigma(\alpha_r)} \in A \cup \{\lightning\}$. From Lemma 3.3.10 we conclude that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}))))) \xrightarrow{\sigma(\alpha_r)} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\vec{\iota}_{|W}))))))$$

By IND it holds that $\vec{g}_{r|W} = \vec{\iota}_{|W}$. Finally, by definition it follows that $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

- Case $r \in K_W$. Follows the same line of reasoning as for case $r \in K_V$.
- Case $r \in I \setminus (K_V \cup K_W)$. This is the more complex situation sketched in Figure 3.4. From requirement SYN we obtain $r \in J_V \cap J_W$. We derive the following transitions using requirement ORI. First, from $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma(c_r^W)]\!]$ it follows that both:

$$P_V(\vec{\iota}_{|V}) \xrightarrow{[\![\sigma(\alpha_r^V|\mathsf{sync}_r^V(\vec{h}_r^V))]\!]} P_V(\sigma(\vec{g}_{r|V}))$$
$$\text{and } P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha_r^W|\mathsf{sync}_r^W(\vec{h}_r^W))]\!]} P_W(\sigma(\vec{g}_{r|W}))$$

Furthermore, $[\![\sigma(\alpha_r)]\!] = [\![\sigma(\alpha_r^V|\alpha_r^W)]\!]$ and by rule PAR we then derive:

$$P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}_r^V))|\mathsf{sync}_W(\sigma(\vec{h}_r^W))]\!]} P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))$$

Note that from $[\![\sigma(\vec{h_r^V})]\!] = [\![\sigma(\vec{h_r^W})]\!]$ it follows that:

$$\theta_H(\gamma_C([\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}_r^V))|\mathsf{sync}_W(\sigma(\vec{h}_r^W))]\!])) = [\![\sigma(\alpha_r)]\!]$$

Since $\underline{[\![\sigma(\alpha_r)]\!]} \in A \cup \{\lightning\}$ follows from $\underline{\alpha_r} \in A \cup \{\lightning\}$, Lemma 3.3.10 allows us to derive that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}))))) \xrightarrow{[\![\sigma(\alpha_r)]\!]} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W})))))))$$

Finally, $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W})))))))$.

- Case $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}))))) \xrightarrow{\omega} Q'$. By Lemma 3.3.11 there is an expression $Q \in \mathsf{S}$ such that $Q' = \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$, and a multi-action $\omega' \in \Omega$ such that $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega')))$, $P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{\omega'} Q$ and $\underline{\theta_H(\gamma_C(\omega'))} \in A \cup \{\lightning\}$. There are three cases where a parallel composition results in a transition. Suppose that $P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{\omega'} Q$ is due to:
  - Rule PARL and $P_V(\vec{\iota}_{|V}) \xrightarrow{\omega'} P_V'$, allowing us to derive transition $P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{\omega'} P_V' \parallel P_W(\vec{\iota}_{|W})$. This is the scenario depicted in Figure 3.3 (a), where the initiative to execute an action comes from the composition expression. Pick an arbitrary index $r \in J_V$. Assume that $r \in J_V \setminus K_V$. Then

the action expression contains an action labelled sync_r^V, which means that $\underline{\theta_H(\gamma_C(\omega'))} \notin A \cup \{\wr, \wr\}$. Contradiction.

Hence, we conclude that $r \in K_V$. From $P_V(\vec{t}_{|V}) \xrightarrow{\omega'} P'_V$, we conclude that there must be a substitution σ such that $\sigma(\vec{d}_{|V}) = \vec{t}_{|V}$ for which $[\![\sigma(c_r)]\!]$, $\omega' = [\![\sigma(\alpha_r)|\mathsf{tag}]\!]$ and $P'_V = P_V(\sigma(\vec{g}_{r|V}))$. Let σ be that substitution. Since requirement IND holds, we know that $\vec{g}_{r|W} = \vec{d}_{|W}$. We may therefore conclude that $P(\vec{t}) \xrightarrow{\omega'} P(\sigma(\vec{g}_r))$. Finally, we may also conclude that $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

- Rule PARR and $P_W(\vec{t}_{|W}) \xrightarrow{\omega'} P'_W$. Follows the same line of reasoning as for rule PARL.
- Rule PAR, and $P_V(\vec{t}_{|V}) \xrightarrow{\omega_V} P'_V$ and $P_W(\vec{t}_{|W}) \xrightarrow{\omega_W} P'_W$, allowing us to derive a transition $P_V(\vec{t}_{|V}) \parallel P_W(\vec{t}_{|W}) \xrightarrow{\omega_V+\omega_W} P'_V \parallel P'_W$, where $\omega_V + \omega_W = \omega'$. Observe that this is the case depicted in Figure 3.4. Note that $P_V(\vec{t}_{|V}) \xrightarrow{\omega_V} P'_V$ implies that there is a substitution σ such that $[\![\sigma(c_r^V)]\!]$ holds, $\omega_V = [\![\sigma(\alpha_r^V|\mathsf{sync}_r^V(\vec{h}_r^V))]\!]$ and $P'_V = P_V([\![\sigma(\vec{g}_{r|V})]\!])$. Likewise, there must be a substitution σ' for which $[\![\sigma'(c_r^W)]\!]$ holds, $\omega_W = [\![\sigma'(\alpha_r^W|\mathsf{sync}_r^W(\vec{h}_r^W))]\!]$ and $P'_W = P_W([\![\sigma'(\vec{g}_{r|W})]\!])$. Let σ and σ' be such substitutions.
From the observation that $\theta_H(\gamma_C(\omega_V + \omega_W)) \in A \cup \{\wr, \wr\}$ holds, it follows that $\gamma_C(\omega_V + \omega_W) = \omega + \underline{\wr\mathsf{sync}_r\wr}$, for some index $r \in I$, because only a single tag is allowed with original action labels. Therefore, it also follows that $r \in I \setminus (K_V \cup K_W)$ and $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$ holds.
From requirement COM it follows that there is a substitution ρ such that $[\![\rho(c_r)]\!]$ holds and $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$. We conclude that $P(\vec{t}) \xrightarrow{\omega} P(\rho(\vec{g}_r))$.
Furthermore, $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$ and therefore by Lemma 3.1.7 it follows that both:

$$P_V(\sigma(\vec{g}_{r|V})) \leftrightarroweq P_V(\rho(\vec{g}_{r|V}))$$
$$\text{and } P_W(\sigma'(\vec{g}_{r|W})) \leftrightarroweq P_W(\rho(\vec{g}_{r|W}))$$

By the congruence of strong bisimilarity with respect to S we obtain that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma'(\vec{g}_{r|W})))))) \leftrightarroweq$$
$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$$

Finally, $P(\rho(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$.

It follows that $[\![P(\vec{t}_{|V})]\!] \leftrightarrows [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{t}_{|V}) \parallel P_W(\vec{t}_{|W})))))]\!]$ from Proposition 3.3.9. □

3.4 State Invariants

The separation tuples inducing the decomposition obtained in Example 3.3.5 are indeed a cleave as shown in Example 3.3.7, but this is by no means the only cleave for process Machine. For instance, also the decomposition we obtained in Example 3.3.1 can be achieved by means of a cleave. The infinite branching of $\text{Machine}_W(\mathsf{false})$ in that example is, however, problematic for the purpose of compositional minimisation. While in this case, as shown by Example 3.3.5, we could avoid the infinite branching of $\text{Machine}_W(\mathsf{false})$ by reducing the amount of synchronisation, this might not always be possible.

Another way to restrict the behaviour of the components is to strengthen the condition expressions of each summand, thus limiting the number of outgoing transitions. We show that so-called *(inductive) state invariants* [9] can be used for this purpose. These state invariants are typically formulated by the user based on the understanding of the modelled behaviour.

Definition 3.4.1. Given an LPE $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$. A Boolean expression ψ such that $\mathsf{FV}(\psi) \subseteq \mathsf{Vars}(\vec{d})$ is called a *state invariant* iff the following holds: for all $i \in I$ and substitutions σ satisfying $[\![\sigma(c_i \wedge \psi)]\!]$, also $[\![\sigma[\vec{d} \leftarrow \sigma(\vec{g}_i)](\psi)]\!]$ should hold.

The essential property of a state invariant is that whenever it holds for some given state it is guaranteed to hold for all states reachable from that state. This follows relatively straightforward from its definition. Next, we define a *restricted* LPE where (some of) the condition expressions are strengthened with a Boolean expression.

Definition 3.4.2. Given an LPE $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$, a Boolean expression ψ such that $\mathsf{FV}(\psi) \subseteq \mathsf{Vars}(\vec{d})$ and a set of indices $J \subseteq I$. We define the restricted LPE, denoted by $P^{\psi,J}$, as follows:

$$\begin{aligned} P^{\psi,J}(\vec{d}:\vec{D}) = &\mathop{+}_{i\in J}\sum_{e_i:E_i} c_i \wedge \psi \to \alpha_i \,.\, P^{\psi,J}(\vec{g}_i) \\ &+ \mathop{+}_{i\in (I\setminus J)}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P^{\psi,J}(\vec{g}_i) \end{aligned}$$

Note that if the Boolean expression ψ in Definition 3.4.2 is a state invariant for the given LPE then for all closed expressions $\vec{t}:\vec{D}$ and substitutions σ for which

$[\![\sigma[d \leftarrow \iota](\psi)]\!]$ holds, it holds that $P(\vec{\iota}) \leftrightarrows P^{\psi,J}(\vec{\iota})$, for any $J \subseteq I$. Therefore, we can use a state invariant of an LPE to strengthen all of its condition expressions.

Moreover, a state invariant of the original LPE can *also* be used to restrict the behaviour of the components obtained from a cleave, as formalised in the following theorem. Note that the set of indices is used to only strengthen the condition expressions of summands that introduce synchronisation, because the condition expressions of independent summands cannot contain the other parameters as free variables. Furthermore, the restriction can be applied to independent summands before the decomposition. The theorem below states that the validity of the decomposition does not change by strengthening the components (induced by separation tuples) using state invariants.

Theorem 3.4.3. Let $P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE with the separation tuples $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ for P. Let ψ be a state invariant of P. For every closed expression $\vec{\iota} : \vec{D}$ and substitution σ for which $[\![\sigma[d \leftarrow \iota](\psi)]\!]$ holds, the following expression, where $C = J^V \cap J^W$, is a valid decomposition:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K_V \cup K_W)\}}(\tau_{\{\mathsf{sync}^i \mid i \in I\}}(\Gamma_{\{\mathsf{sync}^i_V \mid \mathsf{sync}^i_W \to \mathsf{sync}^i \mid i \in I\}}(P_V^{\psi,C}(\vec{\iota}_{|V}) \parallel P_W^{\psi,C}(\vec{\iota}_{|W})))))$$

Proof. Let R be the relation where $P(\vec{\iota'})\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\iota'}_{|V}) \parallel P_W^{\psi,C}(\vec{\iota'}_{|W})))))$ for closed expression $\vec{\iota'}$ exactly when $[\![\sigma[d \leftarrow \vec{\iota'}](\psi)]\!]$ holds for some substitution σ. We show that R is a strong bisimulation relation up to $\leftrightarrows$.

Pick a pair $P(\vec{\iota})\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\iota}_{|V}) \parallel P_W^{\psi,C}(\vec{\iota}_{|W})))))$.

- Case $P(\vec{\iota}) \xrightarrow{\omega} Q'$. Then there is an index $r \in I$ and substitution σ for which $\sigma(\vec{d}) = \vec{\iota}$ such that $[\![\sigma(c_r)]\!]$, $\omega = [\![\sigma(\alpha_r)]\!]$ and $Q' = P(\sigma(\vec{g}_r))$. Furthermore, we know that $[\![\sigma(\psi)]\!]$ holds by definition of R, and, hence also $[\![\sigma[d \leftarrow \sigma(\vec{g}_r)](\psi)]\!]$. There are three cases to consider based on the index r.
  In case $r \in I \setminus (K_V \cup K_W)$, which, because of condition SYN, implies that $r \in (J_V \cap J_W)$, we can conclude that $[\![\sigma(c_r^V \wedge c_r^W \wedge \psi)]\!]$ holds. Therefore, using the same arguments as in the proof of Theorem 3.3.12 for this case, we can deduce that the composition of the two processes can match that ω-transition. Therefore, we can conclude that $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\sigma(\vec{g}_{r|V})) \parallel P_W^{\psi,C}(\sigma(\vec{g}_{r|W}))))))$.
  The other cases of r in the proof of Theorem 3.3.12 deal with unrestricted summands since $r \notin C$. Hence, we only need the additional observation that $[\![\sigma[\vec{d} \leftarrow \sigma(\vec{g}_r)](\psi)]\!]$, from which we conclude $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\sigma(\vec{g}_{r|V})) \parallel P_W^{\psi,C}(\sigma(\vec{g}_{r|W}))))))$.

- Case $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W}))))) \xrightarrow{\omega} Q'$. Similarly to the proof of Theorem 3.3.12 we have three cases to consider.
  For the case $P_V^{\psi,C}(\vec{\imath}_{|V}) \xrightarrow{\omega_V} P'_V$ and $P_W^{\psi,C}(\vec{\imath}_{|W}) \xrightarrow{\omega_W} P'_W$ and rule PAR we observe that if $[\![\sigma(c_r^V \wedge \psi)]\!]$ holds then $[\![\sigma(c_r^V)]\!]$ holds as well, and similarly if $[\![\sigma'(c_r^W \wedge \psi)]\!]$ holds, then also $[\![\sigma'(c_r^W)]\!]$ holds. The remainder of the proof proceeds along the lines of the proof of Theorem 3.3.12. Since for the substitution ρ it holds that $[\![\rho(c_r)]\!]$ and $[\![\rho(\psi)]\!]$ by definition of R, it also follows that $[\![\rho[\vec{d} \leftarrow \rho(\vec{g}_r)](\psi)]\!]$ holds. Therefore, we can conclude that $P(\rho(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$.

  Consider the second case where $P_V^{\psi,C}(\vec{\imath}_{|V}) \xrightarrow{\omega'} P'_V$ and rule PARL is used to derive $P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W}) \xrightarrow{\omega'} P'_V \parallel P_W^{\psi,C}(\vec{\imath}_{|W})$. We observe that $r \in (J_V \setminus K_V)$, and hence $r \notin C$, so the condition expression of the restricted LPE is not modified, and, therefore, the proof again follows the same line of reasoning as in the proof of Theorem 3.3.12. Finally, we note that because $[\![\sigma(c_r)]\!]$ holds and $[\![\sigma(\psi)]\!]$ by definition of R, it follows that $[\![\sigma[\vec{d} \leftarrow \sigma(\vec{g}_r)](\psi)]\!]$. Therefore, it also follows that $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$. □

Observe that the predicate $n \leq 3$ is a state invariant of the LPE Machine in Example 3.1.8. Therefore, we can consider the process $\text{Machine}_W^{\psi,I}$ in Example 3.3.1 for the composition expression, which is finite. This would yield two finite components. However, the state space of $\text{Machine}_W^{\psi,I}$ is still larger than that of P_W in Example 3.3.5.

Finally, we remark that the restricted state space contains deadlock states whenever the invariant does not hold. These deadlocks can be avoided by applying the invariant to the update expression of each parameter instead of the parameter itself without affecting the correctness.

3.5 Implementation

While Theorem 3.3.12 and Definition 3.3.6 together provide the conditions that guarantee that a cleave yields a valid decomposition, requirements (ORI) and (COM) of definition 3.3.6 are difficult to ensure (and verify) due to the semantic nature of these requirements. In this section, we show how, in practice, one can cheaply approximate these correctness requirements by means of a static analysis that relies only on the expressions that occur in an LPE.

3.5.1 Computing a Cleave

For the remainder of this section let $P(\vec{d} : \vec{D}) = +_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be the LPE that we analyse. We assume that the indices for the cleave parameters V and W, such that $V \cup W = \{0, \ldots, n\}$, are given by the user. Computing a promising parameter partitioning automatically is left as future work.

The CLEAVE procedure that is defined in Algorithm 7 yields two separation tuples for P such that these form a cleave. Apart from the user-supplied sets of indices V and W, this algorithm takes an additional input M which we will explain later. The algorithm loops over all summands of the given LPE P and for each summand decides whether the summand is independent by checking the conditions for independent summands. If a summand *is* independent (*i.e.*, meets requirement (IND), which can be checked cheaply), and its condition is independent of the parameters of W (which we approximate by checking whether the relevant expressions of the summand do not contain parameters from W), it is added to K^V; if it is independent and its condition is independent of the parameters of V, it is added to K^W. Otherwise, the summand is not independent and we continue to compute the condition, action and synchronisation expressions for this dependent summand in both separation tuples. In order to construct the synchronisation expressions, we compute a set of variables S, which we call *synchronisation variables*, that mirrors the set of parameters of the other component that may occur within relevant expressions in the summand.

Algorithm 7 relies on two subroutines: one routine to split the (multi-)action of a dependent summand over the two components, and one routine to split its condition expression. The routine for splitting a (multi-)action is detailed in Algorithm 8. This algorithm first checks whether the given (multi-)action *only* depends on the parameters of one component (by checking whether only parameters from d_V or d_W occur) and possibly on sum variables (in the set E). If so, then it makes sense to put that action expression in that component. In that case the action expression does not induce any (additional) synchronisation of parameters. There may be cases in which a (multi-)action relies on parameters of both V and W; since in that case it is impossible to make a decent choice, we rely on input M, supplied by the user, to resolve the distribution.

The routine for computing the condition expression of each component is described by Algorithm 9. We assume that the condition expression is of the shape $\bigwedge_{c \in C} c$ where each element c is a *clause* to simplify the analysis. Note that this can always be achieved by preprocessing the expression. Ideally, the clauses are as small as possible. We provide special treatment for clauses that are *equality* conditions, *i.e.*, expressions of the form $h \approx h'$. For these type of conditions it is possible to use the synchronisation vector to ensure that h is equal to h' whenever synchronisation takes place, which can be advantageous over synchronising the dependencies of h or h' if h is closely related to one component and h' to the other component. However, this is not useful

Algorithm 7 Given an LPE $P(\vec{d}:\vec{D}) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$, two sets of indices $V, W \subseteq \mathbb{N}$ and a function specifying user-defined choices $M : I \to 2^{\mathbb{N}}$ returns two separation tuples that are a cleave as defined in Definition 3.3.6.

1: **procedure** CLEAVE(P, V, W, M)
2: $d_V \leftarrow \mathsf{Vars}(\vec{d}_{|V}), d_W \leftarrow \mathsf{Vars}(\vec{d}_{|W})$
3: $K^V, K^W \leftarrow \emptyset, \emptyset$
4: $c^V, c^W, \alpha^V, \alpha^W, \vec{h}^V, \vec{h}^W \leftarrow \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$
5: **for** $i \in I$ **do**
6: **if** $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_i) \subseteq d_V \cup \{e_i\} \wedge \vec{g}_{i|W} = \vec{d}_{|W}$ **then**
7: $K^V \leftarrow K^V \cup \{i\}$
8: **else if** $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_i) \subseteq d_W \cup \{e_i\} \wedge \vec{g}_{i|V} = \vec{d}_{|V}$ **then**
9: $K^W \leftarrow K^W \cup \{i\}$
10: **else**
11: $(\alpha_V, \alpha_W) \leftarrow$ SPLITACTION$(\alpha_i, d_V, d_W, \{e_i\}, M(i))$
12: $S \leftarrow ((\mathsf{FV}(\vec{g}_{i|V}) \cup \mathsf{FV}(\alpha_V)) \setminus d_V) \cup ((\mathsf{FV}(\vec{g}_{i|W}) \cup \mathsf{FV}(\alpha_W)) \setminus d_W)$
13: $(c_V, c_W, \vec{h_V}, \vec{h_W}) \leftarrow$ SPLITCONDITION$(c_i, d_V, d_W, \{e_i\}, S)$
14: $S \leftarrow S \cup ((\mathsf{FV}(c_V) \cup \mathsf{FV}(\vec{h_V})) \setminus d_V) \cup ((\mathsf{FV}(c_W) \cup \mathsf{FV}(\vec{h_V})) \setminus d_W)$
15: $\vec{s} \leftarrow \langle\rangle$
16: **for** $s \in S$ **do**
17: $\vec{s} \leftarrow \vec{s} \triangleleft \langle s \rangle$
18: $c^V \leftarrow c^V \cup \{i : c_V\}, c^W \leftarrow c^W \cup \{i : c_W\}$
19: $\alpha^V \leftarrow \alpha^V \cup \{i : \alpha_V\}, \alpha^W \leftarrow \alpha^W \cup \{i : \alpha_W\}$
20: $\vec{h}^V \leftarrow \vec{h}^V \cup \{i : \vec{h_V} \triangleleft \vec{s}\}, \vec{h}^W \leftarrow \vec{h}^W \cup \{i : \vec{h_W} \triangleleft \vec{s}\}$
21: $J^V \leftarrow (I \setminus K^W) \cup K^V, J^W \leftarrow (I \setminus K^V) \cup K^W$
22:
23: **return** $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V), (W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$

for equality conditions that compare an expression to a constant. Therefore, we also check whether both expressions h and h' contain free variables.

The set of clauses that cannot be dealt with using synchronisation (the set C' in the algorithm) must be distributed over both components. For correctness it would be sufficient to return the set of clauses C', but we can improve on this by weakening the conditions using the routine COMPUTECONDITION, see Algorithm 10. Consider the call to COMPUTECONDITION on line 14 with the clauses C' and a set of parameters d_V, the sum variables E and the synchronisation variables S. The idea of this procedure is to select all clauses in C' that contain (some of the) variables in $d_V \cup E \cup S$. This is useful since for local parameters in d_V we need to keep the conditions anyway and for variables in $S \setminus d_V$ we want to keep conditions that can restrict their possible values as these variables will be introduced as sum variables (which is a non-deterministic choice). However, every selected clause might depend on variables that are not yet in S' (which is initially equal to S), which is why we iterate until we reach a fixed point.

Algorithm 8 Given a multi-action $\alpha = a^0(f^0)|\ldots|a^k(f^k)$ and sets of variables d_V, d_W and E and a set of indices $M \subseteq \mathbb{N}$, SPLITACTION computes the resulting pair of multi-actions that satisfy the property in Lemma 3.5.1.

```
1:  procedure SplitAction(α, d_V, d_W, E, M)
2:      α_V, α_W ← τ, τ
3:      for 0 ≤ i ≤ k do
4:          if FV(f^i) ⊆ (d_V ∪ E) ∧ FV(f^i) ⊈ (d_W ∪ E) then
5:              α_V ← α_V | a^i(f^i)
6:          else if FV(f^i) ⊆ (d_W ∪ E) ∧ FV(f^i) ⊈ (d_V ∪ E) then
7:              α_W ← α_W | a^i(f^i)
8:          else
9:              if i ∈ M then
10:                 α_V ← α_V | a^i(f^i)
11:             else
12:                 α_W ← α_W | a^i(f^i)
13:
14:     return (α_V, α_W)
```

Note that there are cases for which adding all clauses that depend on variables S' in this manner is not necessarily optimal. For instance, if we have an expression $x < y$, for natural numbers x, y, and only y is included in S' then adding x to S' introduces synchronisation for both x and y, whereas with a more careful analysis the synchronisation of x could be avoided. It is unlikely that this issue can be avoided in general, but we can imagine that specific instances can be avoided by analysing the structure of each clause.

3.5.2 Proof of Correctness

We conclude with a proof of correctness for the presented static analysis.

Lemma 3.5.1. Let α be a multi-action, and d_V, d_W and E three sets of variables and $M \subseteq \mathbb{N}$ a set of indices. Let (α_V, α_W) be the pair of (multi-)actions returned by SPLITACTION(α, d_V, d_W, E, M). For all substitutions σ it holds that $[\![\sigma(\alpha_V|\alpha_W)]\!] = [\![\sigma(\alpha)]\!]$.

Proof. Observe that procedure SPLITACTION terminates. The statement follows directly from the fact that every action expression is syntactically put into either α_V or α_W. □

Lemma 3.5.2. Let C be a set of clauses and S a set of variables. Then procedure COMPUTECONDITION(C, S) terminates and the set of clauses C' it returns satisfies $C' \subseteq C$ and for all variables $x \in S$ if there is a clause $c' \in C$ for which $x \in \mathsf{FV}(c')$ then $c' \in C'$.

Algorithm 9 Given a Boolean expression of the shape $\bigwedge_{c\in C} c$ and sets of variables d_V, d_W, E and S, for which $\mathsf{FV}(\bigwedge_{c\in C} c) \subseteq (d_V \cup d_W \cup E \cup S)$. The procedure SPLITCONDITION computes two condition expressions and two synchronisation vectors that satisfy the property in Lemma 3.5.3.

```
1:  procedure SplitCondition(⋀_{c∈C} c, d_V, d_W, E, S)
2:      h⃗^V, h⃗^W ← ⟨⟩
3:      C′ ← C
4:      for c ∈ C do
5:          if c = h ≈ h′ ∧ FV(h) ≠ ∅ ∧ FV(h′) ≠ ∅ then
6:              if FV(h) ⊆ d_V ∧ FV(d′) ⊆ d_W then
7:                  h⃗_V ← h⃗_V ◁ ⟨h⟩
8:                  h⃗_W ← h⃗_W ◁ ⟨h′⟩
9:                  C′ ← C′ \ {c}
10:             else if FV(h) ⊆ d_W ∧ FV(h′) ⊆ d_V then
11:                 h⃗_V ← h⃗_V ◁ ⟨h′⟩
12:                 h⃗_W ← h⃗_W ◁ ⟨h⟩
13:                 C′ ← C′ \ {c}
14:     C_V ← ComputeCondition(C′, d_V ∪ E ∪ S)
15:     C_W ← ComputeCondition(C′, d_W ∪ E ∪ S)
16:     return (⋀_{c∈C_V} c, ⋀_{c∈C_W} c, h⃗_V, h⃗_W)
```

Proof. First of all, observe that this procedure terminates since for the set of variables S', S'' on line 10 it holds that S' and S'' can only grow with elements in $\bigcup_{c\in C} \mathsf{FV}(c)$ this mean that they can only grow by a finite amount. For every clause c added to C' on lines 2 and 8 it holds that $c \in C$, so $C' \subseteq C$.

We need to show that for all variables $x \in S$ if there is a clause $c' \in C$ for which $x \in \mathsf{FV}(c')$ then $c' \in C'$ holds. Finally, if S is empty then this statement holds. Therefore, assume that S is not empty and we show that the statement is a loop invariant of the do-while loop starting on line 4. Upon entry of the loop the statement does not hold, but since it is a do-while loop the body will be executed at least once. Pick an arbitrary variable $x \in S'$ at line 5 then also $x \in S''$. For all $c \in C$ if $x \in \mathsf{FV}(c)$ then $c \in C'$ on the line 10 since $\mathsf{FV}(c) \cap S'' \neq \emptyset$ and the fact that clauses are never removed from C'. Therefore, the statement holds after a single iteration of the while loop. Next, we observe that the statement is maintained in every iteration of the while loop since no elements are removed from C'. □

Lemma 3.5.3. Let $\bigwedge_{c\in C} c$ be a condition and d_V, d_W, E and S four sets of variables such that $\mathsf{FV}(\bigwedge_{c\in C} c) \subseteq (d_V \cup d_W \cup E \cup S)$. Let $(\bigwedge_{c\in C_V} c, \bigwedge_{c\in C_W} c, \vec{h_V}, \vec{h_W})$ be the tuple returned by SPLITCONDITION$(\bigwedge_{c\in C} c, d_V, d_W, E, S)$. For all substitutions σ it holds that $[\![\sigma(\bigwedge_{c\in C} c)]\!]$ iff $[\![\sigma(\bigwedge_{c\in C_V} c)]\!]$ and $[\![\sigma(\bigwedge_{c\in C_W} c)]\!]$ and $[\![\sigma(\vec{h_V})]\!] = [\![\sigma(\vec{h_W})]\!]$.

Algorithm 10 Given a set of Boolean expressions C and a set of synchronised variables S computes, COMPUTECONDITION computes a set subset of conditions that satisfy the property in Lemma 3.5.2.

```
 1: procedure COMPUTECONDITION(C,S)
 2:     C' ← {c ∈ C | FV(c) = ∅}
 3:     S' ← S
 4:     do
 5:         S'' ← S'
 6:         for c ∈ C do
 7:             if FV(c) ∩ S'' ≠ ∅ then
 8:                 C' ← C' ∪ {c}
 9:                 S' ← S' ∪ FV(c)
10:     while S'' ≠ S'
11:
12:     return C'
```

Proof. Pick an arbitrary substitution σ.

$\Longrightarrow$) Assume that $[\![\sigma(\bigwedge_{c\in C} c)]\!]$ holds. Since $C_V \subseteq C$ by Lemma 3.5.2 it follows that $[\![\sigma(\bigwedge_{c\in C_V} c)]\!]$ holds. Similarly, $[\![\sigma(\bigwedge_{c\in C_W} c)]\!]$ holds as well. Furthermore, consider any expression of the form $h \approx h' \in C$ for which both $\mathsf{FV}(h) \neq \emptyset$ and $\mathsf{FV}(h') \neq \emptyset$ and also $\mathsf{FV}(h) \subseteq d_V$ and $\mathsf{FV}(h') \subseteq d_W$. Then from $[\![\sigma(\bigwedge_{c\in C} c)]\!]$ it follows that $[\![\sigma(h)]\!] = [\![\sigma(h')]\!]$. Similarly, whenever $\mathsf{FV}(h) \subseteq d_W$ and $\mathsf{FV}(h') \subseteq d_V$ it holds that $[\![\sigma(h')]\!] = [\![\sigma(h)]\!]$. Therefore, it follows that $[\![\sigma(\vec{h_V})]\!] = [\![\sigma(\vec{h_W})]\!]$.

$\Longleftarrow$) Assume that $[\![\sigma(\bigwedge_{c\in C_V} c)]\!]$ and $[\![\sigma(\bigwedge_{c\in C_W} c)]\!]$ and $[\![\sigma(\vec{h^V})]\!] = [\![\sigma(\vec{h^W})]\!]$ hold. For all clauses of the form $h \approx h' \in C$ for which $h \approx h' \notin (C_V \cup C_W)$ we know that $[\![\sigma(h)]\!] = [\![\sigma(h')]\!]$ because $[\![\sigma(\vec{h^V})]\!] = [\![\sigma(\vec{h^W})]\!]$ and by the construction of these vectors. Furthermore, by Lemma 3.5.2 we know that for all variables $x \in (d_V \cup E \cup S)$ if there is a clause $c \in C'$ for which $x \in \mathsf{FV}(c)$ then $c \in C_V$. Similarly, for variables $x \in (d_W \cup E \cup S)$ if there is a clause $c \in C'$ for which $x \in \mathsf{FV}(c)$ then $c \in C_W$. Finally, for all $c \in C'$ for which $\mathsf{FV}(c) = \emptyset$ it holds that $c \in (C_V \cap C_W)$. Since $\mathsf{FV}(\bigwedge_{c\in C} c) \subseteq (d_V \cup d_W \cup E \cup S)$ we can conclude that $C_V \cup C_W = C'$. Therefore, from $[\![\sigma(\bigwedge_{c\in C_V} c)]\!]$ and $[\![\sigma(\bigwedge_{c\in C_W} c)]\!]$ it follows that $[\![\sigma(\bigwedge_{c\in C} c)]\!]$. □

Lemma 3.5.4. Let $P(\vec{d} : \vec{D}) = +_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE with two sets of indices $V, W \subseteq \mathbb{N}$ and an indexed set of index sets M. Let tuples $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ be the result of CLEAVE(P, V, W, M); as defined in Algorithm 7. Then these tuples are separation tuples for P as defined in Definition 3.3.2.

Proof. First of all, we will argue that the procedure CLEAVE (Algorithm 7) terminates.

This follows almost directly from the finiteness of I and the finiteness of all expressions in the LPE. The procedures SPLITACTION and SPLITCONDITION terminate by Lemmas 3.5.1 and 3.5.3.

Next, we show that $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ are indeed separation tuples according to Definition 3.3.2. Due to line 21 it follows that $K^V \subseteq J^V$ and c^V, α^V and $\vec{h}^V$ are all indexed sets over $J^V \setminus K^V$. For all indices $i \in (J^V \setminus K^V)$ we observe that SPLITACTION (Algorithm 8) indeed yields two action expressions for which $\mathsf{FV}(\alpha_i^V) \subseteq \mathsf{FV}(\alpha_i)$. Also SPLITCONDITION (Algorithm 9) returns two Boolean conditions for which $\mathsf{FV}(c_i^V) \subseteq \mathsf{FV}(c_i)$ due to Lemma 3.5.3 and two synchronisation expressions for which $\mathsf{FV}(\vec{h}_i^V) \subseteq \mathsf{FV}(c_i)$. Finally, it holds that $S \subseteq (\mathsf{Vars}(\vec{d}) \cup \{e_i\})$ after line 12 since P is an LPE and after line 14 due to the observation that $\mathsf{FV}(c_i^V) \subseteq \mathsf{FV}(c_i)$ and $\mathsf{FV}(\vec{h}_i^V) \subseteq \mathsf{FV}(c_i)$ (and similarly for c_i^W and $\vec{h}_i^W$).

Therefore, for all $i \in (J^V \setminus K^V)$ it holds that $\mathsf{FV}(c_i^V) \cup \mathsf{FV}(\alpha_i^V) \cup \mathsf{FV}(\vec{h}_i^V) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$ on line 20. Finally, for all $i \in K^V$ it holds that $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_{i|V}) \subseteq \mathsf{Vars}(\vec{d}_{|V}) \cup \{e_i\}$ due to line 6. Thus $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ is a separation tuple. The same arguments can be used to show that the tuple $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ is a separation tuple. □

Theorem 3.5.5. Let $P(\vec{d} : \vec{D}) = \sum_{i \in I} \sum_{e_i : E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE with two sets of indices $V, W \subseteq \mathbb{N}$ and an indexed set of index sets M. Let $(V, K^V, J^V, c^V, \alpha^V, \vec{h}^V)$ and $(W, K^W, J^W, c^W, \alpha^W, \vec{h}^W)$ be the separation tuples returned by CLEAVE(P, V, W, M); as defined in Algorithm 7. Then these separation tuples are a cleave as defined in Definition 3.3.6.

Proof. We verify the requirements of Definition 3.3.6. First of all, requirement SYN holds trivially and requirement IND also holds due to line 6 and 8. Next, we show that requirements ORI and COM are satisfied. Pick an arbitrary index $r \in (J^V \cap J^W)$.

- Case requirement ORI. Pick a substitution σ satisfying $[\![\sigma(c_r)]\!]$. Then $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma(c_r^W)]\!]$ hold, and also $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma(\vec{h}_r^W)]\!]$ by Lemma 3.5.3 and the construction of the separation tuples. Furthermore, $[\![\sigma(\alpha_r^V | \alpha_r^W)]\!]$ is equal to $[\![\sigma(\alpha_r)]\!]$ by Lemma 3.5.1.
- Case requirement COM. Consider two subsitutions σ and σ' satisfying $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma'(c_r^W)]\!]$ and $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$. Furthermore, let ρ be a substitution such that $[\![\rho(\vec{d}_{|V})]\!] = [\![\sigma(\vec{d}_{|V})]\!]$ and $[\![\rho(\vec{d}_{|W})]\!] = [\![\sigma'(\vec{d}_{|W})]\!]$.
  First, we show that the synchronisation of variables in S on line 16 of Algorithm 7 put the necessary restrictions on σ and σ'. For every variable $x \in S$ we know from $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$ and the construction on line 16

that $[\![\sigma(x)]\!] = [\![\sigma'(x)]\!]$. Note that the order of insertion into $\vec{s}$ does not matter since the variables are added to both $\vec{h}_r^W$ and $\vec{h}_r^W$ in the same order. This means that $[\![\rho(x)]\!] = [\![\sigma(x)]\!]$ (and thus equal to $[\![\sigma'(x)]\!]$).
From $(\mathsf{FV}(c_r^V) \setminus d_V) \subseteq S$ it follows that $[\![\sigma(c_r^V)]\!] = [\![\rho(c_r^V)]\!]$. The same argument applies to c_r^W for which $[\![\sigma(c_r^W)]\!] = [\![\rho(c_r^W)]\!]$. Furthermore, since $\mathsf{FV}(\vec{h}_r^V) \subseteq d_V$ and $\mathsf{FV}(\vec{h}_r^W) \subseteq d_W$ thus $[\![\sigma(\vec{h}_r^V)]\!] = [\![\rho(\vec{h}_r^V)]\!]$ and $[\![\sigma(\vec{h}_r^W)]\!] = [\![\rho(\vec{h}_r^W)]\!]$. Therefore, we can also conclude that $[\![\rho(c_r)]\!]$ holds since all equality conditions are satisfied by the synchronisation.
Finally, from $(\mathsf{FV}(\alpha_r^V) \setminus d_V) \subseteq S$ and $(\mathsf{FV}(\alpha_r^W) \setminus d_W) \subseteq S$ it also follows that $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r^V|\alpha_r^W)]\!]$, which by Lemma 3.5.1 is equal to $[\![\rho(\alpha_r)]\!]$. Similarly, by $(\mathsf{FV}(\vec{g}_{r|V}) \setminus d_V) \subseteq S$ it follows that $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$ and the same for $[\![\sigma(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$. □

3.6 Case Studies

We have implemented our prototype based on the previously described algorithms to carry out several experiments using specifications written in the high-level language mCRL2 [63], a process algebra generalising the one of Section 3.1.3. To apply the decomposition technique we use the LPEs that the mCRL2 toolset [23] generates as part of the pre-processing step that the toolset performs before further analyses of the specifications are conducted. We compare the results of the monolithic exploration and the exploration based on the decomposition technique. The sources for these experiments can be obtained from the downloadable artifact [92].

Since the decomposition technique is not fully automated yet and several aspects are left as future work we use the practical examples to demonstrate the effectiveness of the technique and point out interesting observations that could be used to improve the cleave algorithm and eventually lead to a completely automated approach. In these case studies we did not manually choose the action label splitting; which means that input M in Algorithm 7 is always the empty function. Therefore, all actions are generated in the W-component when the best choice cannot be made. Furthermore, we only specified an invariant when it is explicitly stated. All experiments are performed on a laptop with an Intel Core i7-7700HQ CPU and 32GB main memory. The only exception is the connect four experiment, which has been performed on a machine with an Intel Xeon Gold 6136 CPU and 15TB main memory.

3.6.1 Alternating Bit Protocol

The alternating bit protocol (ABP) is a communication protocol that uses a single control bit, which is sent along with the message, to implement a reliable communication

channel over two unreliable channels [63]. The specification contains four processes: one for the sender, one for the receiver and two for the unreliable communication channels.

First, we choose the partitioning of the parameters such that one component (ABP_V) contains the parameters of the sender and one communication channel, and the other component (ABP_W) contains the parameters of the receiver and the other communication channel. See Table 3.1 for details concerning their state spaces. We observe that component ABP_V is already larger than the original state space and that it cannot be minimised further, illustrating that traditional compositional minimisation is, in this case, not particularly useful. The composition of the minimised components $\mathsf{ABP}_V || \mathsf{ABP}_W$ is shown under $\mathsf{ABP}_{V||W}$. This shows that it is possible to derive a (slightly) smaller state space.

Table 3.1: Metrics for the alternating bit protocol.

Model	original		minimised	
	#states	#trans	#states	#trans
ABP	182	230	48	58
ABP_V	204	512	204	512
ABP_W	64	196	60	192
$\mathsf{ABP}_{V\|\|W}$	172	220	48	58
ABP_V^{Ψ}	52	90	22	44
ABP_W^{Ψ}	22	44	20	42
$\mathsf{ABP}_{V\|\|W}^{\Psi}$	172	220	48	58
ABP'_V	5	35	5	35
ABP'_W	78	126	28	46
$\mathsf{ABP}'_{V\|\|W}$	76	90	48	58

The main reason for this disappointing result is because the behaviour of each process heavily depends on the state of the other processes, resulting in large components, as this information is lost in the decomposition. We can encode such global information as a state invariant based on the *control flow* parameters (see [114] for a formal definition of the notion of a control flow parameter). The second cleave ($\mathsf{ABP}_V^{\Psi} \parallel \mathsf{ABP}_W^{\Psi}$) for the same parameter partitioning is obtained by restricting the components using this invariant. This does yield a useful decomposition as the state spaces of these components are both smaller than the original state space, even though their composition has again a state space that is only fractionally smaller than that of the monolithic LPE. Finally, we have obtained a cleave into components ABP'_V and ABP'_W where the partitioning is not based on the original processes. Here we have

a process ABP'_V for the data parameter that contains the message that is being sent and all other parameters are in component ABP'_W. This yields a very effective cleave as shown in Table 3.1. However, this example is so small that there is hardly any difference in execution times, which are less than a second.

3.6.2 Decomposing Monolithic Processes

The Chatbox specification [120] describes a chat room facility in which four users can join, leave and send messages. This specification is interesting because it is described as a monolithic process, which means that compositional minimisation is not applicable in the first place. There are (at most) four users in the chatbox, so we can perform a cleave where the behaviour of one user is separated from the behaviour of the other users.

In Table 3.2 we use $\mathsf{Chatbox}_0$ to indicate the component for user 0 and $\mathsf{Chatbox}_{123}$ for the component containing users 1, 2 and 3. The size of the components ($\mathsf{Chatbox}_0$ and $\mathsf{Chatbox}_{123}$) before and after minimisation modulo strong bisimulation are presented to show that these are small and can be further reduced. Furthermore, their composition ($\mathsf{Chatbox}_{0123}$) shows that indeed the decomposition technique can be used quite successfully, because the result under exploration is much smaller than the original state space. Since component $\mathsf{Chatbox}_{123}$ contains (at most) three users we can also apply another cleave to obtain the components $\mathsf{Chatbox}'_1$ and $\mathsf{Chatbox}'_{23}$. In the implementation we only need to add prefixes to the synchronisation and tag action labels since these may not already occur in the LPE. After this computing the composition $\mathsf{Chatbox}'_{123}$ shows that this is even more efficient than computing $\mathsf{Chatbox}_{123}$ directly.

However, we can improve upon these results even further by adapting the specification. Inspecting the Chatbox specification more carefully reveals that many summands of the LPE have disjunctive enabling conditions. These lead to additional synchronisation since the condition cannot be split easily into the two components. We have manually adapted the Chatbox to a strongly bisimilar variant $\mathsf{Chatbox}*$ which avoids these disjunctive enabling conditions. The resulting decomposition shown in Table 3.2 shows that the amount of synchronisation transitions can be greatly reduced this way.

Another example of a monolithic process is the Connect Four specification, which models the behaviour of a game played by two players on a board with seven columns and four rows. Using the decomposition procedure we first obtain a component for the left-most column and a component for the six remaining columns. Next, we apply the decomposition to the process for the six remaining columns recursively until we have one component for every column. In Table 3.2 we can see the state space of the process for only column seven. We have left out the state spaces of the other components for a single column since these are all similar in size. Then we compose

columns six and seven, which is the state space listed under Connect Four$_{67}$ and the composition of column five, six and seven is listed under Columns Connect Four$_{5-7}$, *etcetera*. Repeating this process until we have composed all the columns shows that we can obtain a state space that is roughly half in size compared to the original state space in the number of states and transitions.

Table 3.2: State space metrics for the chatbox [120] and a connect four specification.

Model	exploration		minimised	
	#states	#transitions	#states	#transitions
Chatbox	65 536	2 621 440	16	144
Chatbox$_0$	128	4 352	128	3 456
Chatbox$_{123}$	512	37 888	8	440
Chatbox$_{0\|\|123}$	1 024	22 528	16	144
Chatbox$'_1$	32	1 344	16	1 120
Chatbox$'_{23}$	16	1 280	4	276
Chatbox$'_{1\|\|23}$	128	8 192	16	144
Chatbox$*$	65 536	720 896	16	144
Chatbox$*_0$	128	768	128	768
Chatbox$*_1$	32	224	32	224
Chatbox$*_{23}$	16	216	4	52
Chatbox$*_{1\|\|23}$	128	1 920	8	104
Chatbox$*_{0\|\|123}$	1 024	11 776	16	144
Connect Four	4 571 392 011	18 814 446 993	418 390 653	2 079 589 075
Connect Four$_7$	31	1 664	31	1 664
Connect Four$_{67}$	961	40 992	961	40 992
Connect Four$_{5...7}$	29 791	908 876	23 327	713 264
Connect Four$_{4...7}$	723 137	13 059 584	503 723	9 158 684
Connect Four$_{3...7}$	15 615 413	204 751 466	13 560 351	180 369 650
Connect Four$_{2...7}$	420 370 881	4 541 332 512	326 297 880	3 628 882 674
Connect Four$_{1...7}$	2 388 678 550	10 967 818 533	418 390 653	2 079 589 075

3.6.3 Practical Specifications

The Register specification [77] describes a wait-free handshake register and the WMS specification is a workload management system [118], used at CERN. For these

two experiments we found that partitioning the parameters into a set of control flow parameters and remaining parameters yields good results. For WMS we observed that one control flow parameter was only used in the initialisation part of the process. This meant that splitting it off from the data parameters caused this initialisation to become possible in every state, leading to many unnecessary synchronisation transitions. Therefore, we have also performed an alternative cleave into components WMS'_V and WMS_W, whose components are (much) smaller.

Table 3.3: State space metrics for Hesselink's wait-free handshake register [77] and a workload management system used at CERN [118].

Model	exploration		minimised	
	#states	#transitions	#states	#transitions
Register	914 048	1 885 824	1 740	3 572
$\mathsf{Register}_V$	464	10 672	464	10 672
$\mathsf{Register}_W$	97 280	273 408	5 760	16 832
$\mathsf{Register}_{V\|\|W}$	76 416	157 952	1 740	3 572
WMS	155 034 776	2 492 918 760	44 526 316	698 524 456
WMS_V	212 992	5 144 576	212 992	2 801 664
WMS_W	1 903 715	121 945 196	414 540	26 429 911
$\mathsf{WMS}_{V\|\|W}$	64 635 040	1 031 080 812	44 526 316	698 524 456
WMS'_V	311 296	7 159 808	294 912	6 815 744
WMS'_W	345 527	26 084 118	75 121	5 665 871
$\mathsf{WMS}'_{V\|\|W}$	64 635 040	1 049 716 700	44 526 316	698 524 456

3.6.4 Execution Times

We also consider the total execution time and maximum amount of memory required to obtain the original state space using exploration and the state space obtained using the decomposition technique. The execution times in **s**econds or **h**ours required to obtain the state space under 'exploration' in Tables 3.2 and 3.3, excluding the final minimisation step of the original or composition state space which are only shown for reference. The cost of the static analysis of the cleave itself was in the range of several milliseconds.

Although in most cases the decomposition improves both runtime and peak memory usage this does not hold for the Connect Four specification. Here, we observe that the peak memory usage is much higher. This memory peak is caused by the strong

Table 3.4: Execution times and maximum memory usage measurements.

Model	monolithic		decomposition	
	time	memory	time	memory
Chatbox	4.76s	21.9MB	0.2s	15.7MB
Register	7.94s	99.7MB	1.56s	47.7MB
WMS	2.4h	15.1GB	0.8h	11.8GB
WMS′	2.4h	15.1GB	0.6h	3.0GB
Connect Four	25h	437GB	20h	543GB

bisimulation minimisation step for the process for columns two to seven. Although this seems worse at first we must note that the final minimisation step requires far more memory and time, whereas with the decomposition we obtain a state space that is almost half in size of monolithic exploration directly.

3.7 Conclusion

We have presented a decomposition technique, referred to as cleave, that can be applied to any monolithic process with the structure of an LPE and have shown that the result is always a valid decomposition. Furthermore, we have shown that state invariants can be used to improve the effectiveness of the decomposition. We consider defining a static analysis to automatically derive the parameter partitioning for the practical application of this technique as future work. Furthermore, the cleave is currently not well-suited for applying the typically more useful abstraction based on (divergence-preserving) branching bisimulation minimisation [56]. The reason for this is that τ-actions are 'decorated' with synchronisation actions and tags. As a result these actions become visible, and therefore effectively branching bisimilarity yields the same reduction as strong bisimilarity.

It seems that the way communication is formalised in the process algebra that we consider is essential to achieve a valid (*i.e.*, strong bisimulation preserving) decomposition based on the data parameters. In any process algebra in which the synchronised parallel composition between processes (similar to our rule Par) is renamed into a single invisible action, often also denoted by τ, such as CCS [103] and CSP [78] it is impossible for a parallel composition of two components to mimic a visible transition of the monolithic process that requires synchronisation of data since we cannot distinguish which transition of the monolithic process it belongs to, and therefore preserve strong bisimulation. If communication does result in a visible transition then

an expressive *relabelling* operator could be used to rename the resulting communication. Here, expressive means that it must for example be able to ignore the action data parameters that are used purely to synchronise data, but also rename actions to τ. However, such a relabelling operator is often not present in process algebras, because it does not behave nicely with respect to congruence. Finally, in the presence of global variables the synchronisation between components could easily be achieved, but then the question is how the state space of individual components can be derived. However, finding the exact minimal set of features required for a process algebra to achieve a cleave could be an interesting future direction.

On a more practical note, the main disadvantage of the current implementation is that the parameters must be provided by the user, and this requires quite some insight into the specification. As proposed earlier possible future work would be to employ the techniques in [114] to analyse the dependencies in order to automatically find a decent partitioning. Furthermore, we could use information about the original process specification from which the monolithic process is derived to guide the partitioning. In its current form the technique is difficult to apply to systems with infinite data sorts since manual analysis is required to preserve the finiteness of the components. The optimal parameter partitioning is also an algorithmic optimisation problem since the choice for the partition also influences the sizes of the intermediate state spaces.

Another disadvantage of the current technique is that only specific summands of the components can synchronise with each other due to the unique indices assigned to the synchronisation actions. However, some early experiments suggest that it might be more efficient to allow different summands of the components to synchronise with each other whenever different summands of the monolithic LPE deal with the same action labels. Finally, the technique could be extended to allow splitting into more than two components, yielding synchronisation over multiple components simultaneously. This is also related to the problem of nested applications of the cleave procedures.

Chapter 4

On-The-Fly Solving for Symbolic Parity Games

A parity game is a two-player game with an ω-regular winning condition, played by players $\Diamond$ ('even') and $\Box$ ('odd') on a directed graph. The true complexity of solving parity games is still a major open problem, with the most recent breakthroughs yielding algorithms running in quasi-polynomial time, see, *e.g.*, [86, 25]. Apart from their intriguing status, parity games pop up in various fundamental results in computer science (*e.g.*, in the proof of decidability of a monadic second-order theory). In practice, parity games provide an elegant, uniform framework to encode many relevant decision problems, which include model checking problems, synthesis problems and behavioural equivalence checking problems.

Often, a decision problem that is encoded as a parity game, can be answered by determining which of the two players wins a designated vertex in the game graph. Depending on the characteristics of the game, it may be the case that only a fraction of the game is relevant for deciding which player wins a vertex. For instance, deciding whether a transition system satisfies an invariant can be encoded by a simple, solitaire (*i.e.*, single player) parity game. In such a game, player $\Box$ wins all vertices that are sinks (*i.e.*, have no successors), and all states leading to such sinks, so checking whether sinks are reachable from a designated vertex suffices to determine whether this vertex is won by $\Box$, too. Clearly, as soon as a sink is detected, any further inspection of the game becomes irrelevant.

A complicating factor is that in practice, the parity games that encode decision problems are not given explicitly. Rather, they are specified in some higher-order logic such as a parameterised Boolean equation system, see, *e.g.* [33]. Exploring the

parity game from such a higher-order specification is, in general, time-and memory-consuming. To counter this, symbolic exploration techniques have been proposed, see *e.g.* [88]. These explore the game graph on-the-fly and exploit efficient symbolic data structures such as LDDs [41] to represent sets of vertices and edges. Many parity game solving algorithms can be implemented quite effectively using such data structures [89, 124, 126], so that in the end, exploring the game graph often remains the bottleneck.

In this chapter, we study how to combine the exploration of a parity game and the on-the-fly solving of the explored part, with the aim to speed-up the overall solving process. The central problem when performing on-the-fly solving during the exploration phase is that we have to deal with incomplete information when determining the winner for a designated vertex. Moreover, in the symbolic setting, the exploration order may be unpredictable when advanced strategies such as *chaining* and *saturation* [29] are used.

To formally reason about all possible exploration strategies and the artefacts they generate, we introduce the concept of an *incomplete parity game*, and an ordering on these. Incomplete parity games are parity games where for some vertices not all outgoing edges are necessarily known. In practice, these could be identified by, *e.g.*, the *todo* queue in a classical breadth-first search. The extra information captured by an incomplete parity game allows us to characterise the *safe* set for a given player α. This is a set of vertices for which it can be established that if player α wins the vertex, then she cannot lose the vertex if more information becomes available. We prove an optimality result for safe sets, which, informally, states that a safe set for player α is also the largest set with this property (see Theorem 4.2.12).

The vertices won by player α in an α-safe set can be determined using a standard parity game solving algorithm such as, *e.g.*, Zielonka's recursive algorithm [140] or Priority Promotion [4]. However, these algorithms may be less efficient as on-the-fly solvers. For this reason, we study three symbolic *partial* solvers: *solitaire winning cycle* detection, *forced winning cycle* detection and *fatal attractors* [81]. In particular cases, first determining the safe set for a player and only subsequently solving the game using one of these partial solvers will incur an additional overhead. As a final result, we therefore prove that all these solvers can be (modified to) run on the incomplete game as a whole, rather than on the safe set of a player (see Propositions 4.3.5-4.3.11).

As a proof of concept, we have implemented an (open source) symbolic tool for the mCRL2 toolset [23], that explores a parity game specified by a parameterised Boolean equation system and solves these games on-the-fly. We report on the effectiveness of our implementation on typical parity games stemming from, *e.g.*, model checking and equivalence checking problems, showing that it can speed up the process with several orders of magnitude, while adding low overhead if the entire game is needed for solving.

Related Work Our work is related to existing techniques for solving symbolic parity games such as [89, 88], as we extend these existing methods with on-the-fly solving. Naturally, our work is also related to existing work for on-the-fly model checking. This includes work for on-the-fly (explicit) model checking of regular alternation-free modal mu-calculus formulas [99] and work for on-the-fly symbolic model checking of RCTL [3]. There are also other works related to these so-called *local* model checking techniques such as [128] and [44]. These present specialised algorithms that attempt to solve the model checking problem by (lazily) exploring as little as possible from the underlying model.

Compared to these our method is more general as it can be applied to the full modal mu-calculus with data, which subsumes RCTL and the alternation-free subset. Our method is also lazy in a certain sense since the early termination can ensure that we do not explore the complete underlying method. However, compared to the existing methods it is agnostic to the exploration strategy that is employed and the (partial) solvers that are being used. Optimisations such as the observation that checking LTL formulas of type AG reduces to reachability checks [43] are a special case of our methods and partial solvers. Furthermore, our methods are not restricted to model checking problems only and can be applied to any parity game, including decision problems such as equivalence checking [26].

Our idea of incomplete parity games is related to three-valued modal logics as presented in [20]. Here, a partial structure is defined where the solution to the model checking question is also the solution for the complete structure. However, naturally the solution can also be indeterminate when the partial information is not sufficient to make a definite conclusion about the full structure. Our main contribution here is that we define similar concepts for parity games and prove certain optimality results.

Outline In Section 4.1 we recall parity games. In Section 4.2 we introduce incomplete parity games and show how partial solving can be applied correctly. In Section 4.3 we present several partial solvers that we employ for on-the-fly solving. In Section 4.4 we discuss the implementation of these techniques. Finally, in Section 4.5 we apply the presented techniques to several practical examples.

4.1 Preliminaries

A parity game is an infinite-duration, two-player game that is played on a finite directed graph. The objective of the two players, called *even* (denoted by $\Diamond$) and *odd* (denoted by $\Box$), is to win vertices in the graph.

Definition 4.1.1. A *parity game* is a directed graph $G = (V, E, p, (V_\Diamond, V_\Box))$, where

- V is a finite set of vertices, partitioned in sets $V_\Diamond$ and $V_\Box$ of vertices owned by $\Diamond$ and $\Box$, respectively;
- $E \subseteq V \times V$ is the edge relation;
- $p : V \to \mathbb{N}$ is a function that assigns a *priority* to each node.

Henceforth, let $G = (V, E, p, (V_\Diamond, V_\Box))$ be an arbitrary parity game. Throughout this chapter, we use α to denote an arbitrary player and $\bar{\alpha}$ denotes the opponent. We write vE to denote the set of successors $\{w \in V \mid (v,w) \in E\}$ of vertex v. The set $\mathsf{sinks}(G)$ is defined as the largest set $U \subseteq V$ satisfying for all $v \in U$ that $vE = \emptyset$; *i.e.*, $\mathsf{sinks}(G)$ is the set of all sinks: vertices without successors. If we are only concerned with the sinks of player α, we write $\mathsf{sinks}_\alpha(G)$; *i.e.*, $\mathsf{sinks}_\alpha(G) = V_\alpha \cap \mathsf{sinks}(G)$. We write $G \cap U$, for $U \subseteq V$, to denote the subgame $(U, (U \times U) \cap E, p\restriction_U, (V_\Diamond \cap U, V_\Box \cap U))$, where $p\restriction_U (v) = p(v)$ for all vertices $v \in U$.

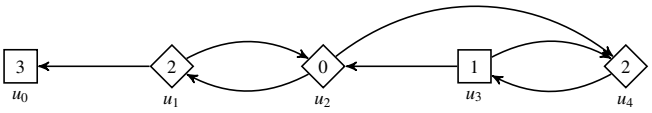


Figure 4.1: An example parity game

Example 4.1.2. Consider the graph depicted in Figure 4.1, representing a parity game. Diamond-shaped vertices are owned by player $\Diamond$, whereas box-shaped vertices are owned by player $\Box$. The priority of a vertex is written inside the vertex. Vertex u_0 is a sink owned by player $\Box$. □

Plays and strategies. The game is played as follows. Initially, a token is placed on a vertex of the graph. The owner of a vertex on which the token resides gets to decide the successor vertex (if any) that the token is moved to next. A maximal sequence of vertices (*i.e.*, an infinite sequence or a finite sequence ending in a sink) visited by the token by following this simple rule is called a *play*. A finite play π is won by player $\Diamond$ if the sink in which it ends is owned by player $\Box$, and it is won by player $\Box$ if the sink is owned by player $\Diamond$. An infinite play π is won by player $\Diamond$ if the minimal priority that occurs infinitely often along π is even, and it is won by player $\Box$ otherwise.

A strategy $\sigma_\alpha : V^* V_\alpha \to V$ for player α is a partial function that prescribes where player α moves the token next, given a sequence of vertices visited by the token. A play $v_0 v_1 \ldots$ is *consistent* with a strategy σ if and only if $\sigma(v_0 \ldots v_i) = v_{i+1}$ for all i for which $\sigma(v_0 \ldots v_i)$ is defined. Strategy σ_α is winning for player α in vertex v iff all plays consistent with σ_α and starting in v are won by α. Player α wins vertex v if and only if she has a winning strategy σ_α for vertex v. The *parity game solving problem*

asks to compute the set of vertices $W_\Diamond$, won by player $\Diamond$ and the set $W_\Box$, won by player $\Box$. Note that since parity games are *determined* [140, 100], every vertex is won by one of the two players. That is, the sets $W_\Diamond$ and $W_\Box$ partition the set V.

Example 4.1.3. Consider the parity game depicted in Figure 4.1. In this game, the strategy $\sigma_\Diamond$, partially defined as $\sigma_\Diamond(\pi u_1) = u_2$ and $\sigma_\Diamond(\pi u_2) = u_1$, for arbitrary π, is winning for player $\Diamond$ in u_1 and u_2. Player $\Box$ wins vertex u_3 using strategy $\sigma_\Box(\pi u_3) = u_4$, for arbitrary π. Note that player $\Diamond$ is always forced to move the token from u_4 to u_3. Vertex u_0 is a sink, owned by player $\Box$, and hence, won by player $\Diamond$. $\Box$

Dominions. A strategy σ_α is said to be *closed* on a set of vertices $U \subseteq V$ iff every play, consistent with σ_α and starting in a vertex $v \in U$ remains in U. If player α has a strategy that is closed on U, we say that the set U is α-closed. A *dominion* for player α, also called α-dominion, is a set of vertices $U \subseteq V$ such that player α has a strategy σ_α that is closed on U and which is winning for α. Note that the sets $W_\Diamond$ and $W_\Box$ are dominions for player $\Diamond$ and player $\Box$, respectively, and, hence, every vertex won by player α must belong to an α-dominion.

Example 4.1.4. Reconsider the parity game of Figure 4.1. Observe that player $\Box$ has a closed strategy on $\{u_3, u_4\}$, which is also winning for player $\Box$. Hence, the set $\{u_3, u_4\}$ is an $\Box$-dominion. For a similar reason, the set $\{u_0, u_1, u_2\}$ is a $\Diamond$-dominion. On the other hand, the set $\{u_2, u_3, u_4\}$ is $\Diamond$-closed. However, none of the strategies for which $\{u_2, u_3, u_4\}$ is closed for player $\Diamond$ is winning for her; therefore $\{u_2, u_3, u_4\}$ is not an $\Diamond$-dominion. $\Box$

Predecessors, control predecessors and attractors. Let $U \subseteq V$ be a set of vertices. We write $\mathsf{pre}(G, U)$ to denote the set of predecessors $\{v \in V \mid \exists u \in U : u \in vE\}$ of U in G. The control predecessor set of U for player α in G, denoted $\mathsf{cpre}_\alpha(G, U)$, contains those vertices for which α is able to *force* entering U in one step. It is defined as follows:

$$\mathsf{cpre}_\alpha(G, U) = (V_\alpha \cap \mathsf{pre}(G, U)) \cup (V_{\bar{\alpha}} \setminus (\mathsf{pre}(G, V \setminus U) \cup \mathsf{sinks}(G)))$$

Note that both pre and cpre are monotone operators on the complete lattice $(2^V, \subseteq)$. The α-attractor to U in G, denoted $\mathsf{Attr}_\alpha(G, U)$, is the set of vertices from which player α can force play to reach a vertex in U:

$$\mathsf{Attr}_\alpha(G, U) = \mu Z.(U \cup \mathsf{cpre}_\alpha(G, Z))$$

The α-attractor to U can be computed by means of a fixed point iteration, starting at U and adding α-control predecessors in each iteration until a stable set is reached. We note that the α-attractor to an α-dominion D is again an α-dominion.

Example 4.1.5. Consider the parity game G of Figure 4.1 once again. The $\Diamond$-control predecessors of $\{u_2\}$ is the set $\{u_1\}$. Note that since player $\Box$ can avoid moving to u_2 from vertex u_3 by moving to vertex u_4, vertex u_3 is not among the $\Diamond$-control predecessors of $\{u_2\}$. The $\Diamond$-attractor to $\{u_2\}$ is the set $\{u_1, u_2\}$, which is the largest set of vertices for which player $\Diamond$ has a strategy to force play to the set of vertices $\{u_2\}$. □

4.2 Incomplete Parity Games

In many practical applications that rely on parity game solving, the parity game is gradually constructed by means of an exploration, often starting from an 'initial' vertex. This is, for instance, the case when using parity games in the context of model checking or when deciding behavioural preorders or equivalences. For such applications, it may be profitable to combine exploration and solving, so that the costly exploration can be terminated when the winner of a particular vertex of interest (often the initial vertex) has been determined. The example below, however, illustrates that one cannot naively solve the parity game constructed so far.

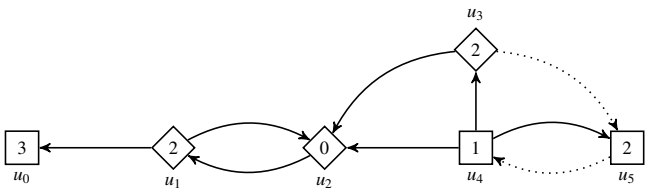


Figure 4.2: A parity game where the dotted edges are not yet known.

Example 4.2.1. Consider the parity game G in Figure 4.2, consisting of all vertices and only the solid edges. This game could, for example, be the result of an exploration starting from u_4. Then $G \cap \{u_0, u_1, u_2, u_3, u_4, u_5\}$ is a subgame for which we can conclude that all vertices form an $\Diamond$-dominion. However, after exploring the dotted edges, player $\Box$ can escape to vertex u_4 from vertex u_5. Consequently, vertices u_4 and u_5 are no longer won by player $\Diamond$ in the extended game. Furthermore, observe that the additional edge from u_3 to u_5 does not affect the previously established fact that player $\Diamond$ wins this vertex. □

To facilitate reasoning about games with incomplete information, we first introduce the notion of an *incomplete* parity game.

Definition 4.2.2. An *incomplete* parity game is a structure $\mathcal{D} = (G, I)$, where G is a parity game $(V, E, p, (V_\Diamond, V_\Box))$, and $I \subseteq V$ is a set of vertices with potentially unexplored successors. We refer to the set I as the set of *incomplete vertices*; the set $V \setminus I$ is the set of *complete vertices*.

Observe that $(G, \emptyset)$ is a 'standard' parity game. We permit ourselves to use the notation for parity game notions such as plays, strategies, dominions, *etcetera*, also in the context of incomplete parity games. In particular, for $\mathcal{D} = (G, I)$, we write $\mathsf{pre}(\mathcal{D}, U)$ and $\mathsf{Attr}_\alpha(\mathcal{D}, U)$ to indicate $\mathsf{pre}(G, U)$ and $\mathsf{Attr}_\alpha(G, U)$, respectively. Furthermore, we define $\mathcal{D} \cap U$ as the structure $(G \cap U, I \cap U)$.

Intuitively, while exploring a parity game, we extend the set of vertices and edges by exploring the incomplete vertices. Doing so gives rise to potentially new incomplete vertices. At each stage in the exploration, the incomplete parity game extends incomplete parity games explored in earlier stages. We formalise the relation between incomplete parity games, abstracting from any particular order in which vertices and edges are explored.

Definition 4.2.3. Let $\mathcal{D} = ((V, E, p, (V_\Diamond, V_\Box)), I)$ and $\mathcal{D}' = ((V', E', p', (V'_\Diamond, V'_\Box)), I')$ be incomplete parity games. We write $\mathcal{D} \sqsubseteq \mathcal{D}'$ iff the following conditions hold:

(1) $V \subseteq V'$, $V_\Diamond \subseteq V'_\Diamond$ and $V_\Box \subseteq V'_\Box$;

(2) $E \subseteq E'$ and $((V \setminus I) \times V) \cap E' \subseteq E$;

(3) $p = p' \restriction_V$;

(4) $I' \cap V \subseteq I$

Conditions (1) and (3) are self-explanatory. Condition (2) states that on the one hand, no edges are lost, and, on the other hand, E' can only add edges from vertices that are incomplete: for complete vertices, E' specifies no new successors. Finally, condition (4) captures that the set of incomplete vertices I' cannot contain vertices that were previously complete. We note the following.

Lemma 4.2.4. The ordering $\sqsubseteq$ is reflexive, anti-symmetric and transitive.

Proof. Reflexivity is trivial. For anti-symmetry we only need the observation that if $A \subseteq B$ and $B \subseteq A$ then $A = B$. We show transitivity, *i.e.*, that $\mathcal{D} \sqsubseteq \mathcal{D}''$ holds, where $\mathcal{D}'' = ((V'', E'', p'', (V''_\Diamond, V''_\Box)), I'')$ and $\mathcal{D} \sqsubseteq \mathcal{D}' \sqsubseteq \mathcal{D}''$. Transitivity of (1) and (3) follows from the transitivity of $\subseteq$ and the definition of $\restriction$. For (2) we have to show that $((V \setminus I) \times V) \cap E'' \subseteq E$, but this follows from the fact that $E' \subseteq E''$ and $((V \setminus I) \times V) \cap E' \subseteq E$. Finally, for (4) show that $I'' \cap V \subseteq I$. Since $I'' \cap V' \subseteq I'$ it follows that $I'' \cap V' \cap V \subseteq I' \cap V$ and thus $I'' \cap V \subseteq I' \cap V$, because $V \subseteq V'$. Furthermore, since $I' \cap V \subseteq I$ it follows that $I'' \cap V \subseteq I$. □

Example 4.2.5. Suppose that $\mathcal{D} = (G, I)$ is the incomplete parity game depicted in Figure 4.2, where G is the game with all vertices and only the solid edges, and $I = \{u_3, u_5\}$. Then $\mathcal{D} \sqsubseteq \mathcal{D}'$, where $\mathcal{D}' = (G', I')$ is the incomplete parity game where G' is the depicted game with all vertices and both the solid edges and dotted edges, and $I' = \emptyset$. □

Let us briefly return to Example 4.2.1. We concluded that the winner of vertex u_4 (and also u_5) changed when adding new information. The reason is that player □ has a strategy to reach an *incomplete* vertex owned by her. Such an incomplete vertex may present an opportunity to escape from plays that would be non-winning otherwise. On the other hand, the incomplete vertex u_3 has already been sufficiently explored to allow for concluding that this vertex is won by player ◇, even if more successors are added to u_3. This suggests that for some vertices, we can decide their winner in an incomplete parity game and preserve that winner in all future extensions of the game. We formally characterise this set of vertices in the definition below.

Definition 4.2.6. Let $\mathcal{D} = (G, I)$, with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game. The α-*safe* vertices for $\mathcal{D}$, denoted by $\mathsf{safe}_\alpha(\mathcal{D})$, is the set $V \setminus \mathsf{Attr}_{\bar{\alpha}}(G, V_{\bar{\alpha}} \cap I)$.

Example 4.2.7. Consider the incomplete parity game $\mathcal{D}$ of Example 4.2.5 once more. We have $\mathsf{safe}_\Diamond(\mathcal{D}) = \{u_0, u_1, u_2, u_3\}$ and $\mathsf{safe}_\Box(\mathcal{D}) = \{u_0, u_1, u_2, u_4, u_5\}$. □

In the remainder of this section, we show that it is indeed the case that while exploring a parity game, one can only safely determine the winners in the sets $\mathsf{safe}_\Box(\mathcal{D})$ and $\mathsf{safe}_\Diamond(\mathcal{D})$, respectively. More specifically, we show (Lemma 4.2.8) that all α-dominions found in $\mathsf{safe}_\alpha(\mathcal{D})$ are preserved in extensions of the game, and (Lemma 4.2.10) the vertices not in $\mathsf{safe}_\alpha(\mathcal{D})$ are not necessarily won by the same player in extensions of the game.

Lemma 4.2.8. Given two incomplete games $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. Any α-dominion in $\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$ is also an α-dominion in $\mathcal{D}'$.

Proof. Let $\mathcal{D} = (G, I)$, with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game, and $\mathcal{D}' = (G', I')$, with $G' = (V', E', p', (V'_\Diamond, V'_\Box))$. Assume that $\mathcal{D} \sqsubseteq \mathcal{D}'$ and suppose that $D \subseteq V$ is an α-dominion in $G \cap \mathsf{safe}_\alpha(\mathcal{D})$. Observe that since $V \subseteq V'$, also $D \subseteq V'$. This means that player α must have a strategy σ that is closed on D and for which every play consisting with that strategy and starting in a vertex in D is winning for α. Let σ be such a strategy. We define strategy σ' as follows. Let $\pi = v_0 \dots v_n \in V^* V_\alpha$ be an arbitrary path through D. Then $\sigma'(\pi) = \sigma(\pi)$ whenever σ is defined for π, and $\sigma'(\pi) = v'$ for an arbitrary $v' \in \{v'' \in D \mid (v_n, v') \in E\} \subseteq \{v'' \in D \mid (v_n, v') \in E'\}$ in case $\{v'' \in D \mid (v_n, v') \in E\} \neq \emptyset$ and σ is not defined for π.

Consider an arbitrary play $v_0 v_1 \ldots$ in G', consistent with σ' and starting in $v_0 \in D$. Suppose that there is some vertex on this play that is not in D. Let v_i be the first such vertex; *i.e.*, $v_j \in D$ for all $j < i$. We distinguish two cases.

- Assume σ is defined for $v_0 v_1 \ldots v_{i-1}$. Then $\sigma(v_0 v_1, \ldots v_{i-1}) \in D$, since σ is closed on D in G. But then also $\sigma'(v_0 v_1 \ldots v_{i-1}) = v_i \in D$. Contradiction.
- Next, assume that σ is not defined for $v_0 v_1 \ldots v_{i-1}$. We distinguish two further cases.
  - Case $v_{i-1} \in V_\alpha$. Since σ is closed on D in G, it must be the case that $\{v'' \in V \mid (v_{i-1}, v'') \in E\} \subseteq D$. By construction, $\sigma'(v_0 v_1 \ldots v_{i-1}) = v_i \in D$. Contradiction.
  - Case $v_{i-1} \in V_{\bar{\alpha}}$. Since σ is closed on D in G, it must be the case that $\{v'' \in V \mid (v_{i-1}, v'') \in E\} \subseteq D$. Furthermore, since $D \subseteq V \cap \mathsf{safe}_\alpha(\mathcal{D})$, $(I' \cap V) \subseteq I$ and $(V \setminus I) \times V) \cap E' \subseteq E$, also $\{v'' \in V' \mid (v_{i-1}, v'') \in E'\} \subseteq D$. But then also $v_i \in D$. Contradiction.

Since all cases lead to a contradiction, we find that $v_i \in D$. But then all plays consistent with σ' must remain in D.

It remains to argue that σ' is a winning play. However, this follows from the fact that all plays π, starting in D and consistent with σ' are also plays consistent with σ. Since σ was a winning strategy for player α, also σ' is winning for α in (G', I'). □

Example 4.2.9. Recall that in Example 4.2.7, we found that $\mathsf{safe}_\Diamond(\mathcal{D}) = \{u_0, u_1, u_2, u_3\}$. Observe that in the incomplete parity game $\mathcal{D}$ of Example 4.2.5, restricted to vertices $\{u_0, u_1, u_2, u_3\}$, all vertices are won by player $\Diamond$, and, hence, $\{u_0, u_1, u_2, u_3\}$ is an $\Diamond$-dominion. Following Lemma 4.2.8 we can indeed conclude that this remains an $\Diamond$-dominion in all extensions of $\mathcal{D}$, and, in particular, for the (complete) parity game $\mathcal{D}'$ of Example 4.2.5. □

Lemma 4.2.10. Let $\mathcal{D}$ be an incomplete parity game. Suppose that W is an α-dominion in $\mathcal{D}$. If $W \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$, then there is an (incomplete) parity game $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$and all vertices in $W \setminus \mathsf{safe}_\alpha(\mathcal{D})$ are won by $\bar{\alpha}$.

Proof. Let $\mathcal{D} = (G, I)$ be an incomplete parity game, with $G = (V, E, p, (V_\Diamond, V_\Box))$, and assume that $W \subseteq V$ is an α-dominion. Suppose that $W \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$.

Let $G' = (V', E', p', (V_\Diamond, V_\Box))$, with $V' = V \cup \{z\}$, for fresh vertex $z \notin V$, $E' = E \cup \{(v, z) \mid v \in I\}$, $p' \restriction_V = p$ and $p'(z) = 0$, and $V'_\alpha = V_\alpha \cup \{z\}$ and $V'_{\bar{\alpha}} = V_{\bar{\alpha}}$. Then it follows that $\mathcal{D} \sqsubseteq \mathcal{D}'$ for $\mathcal{D}' = (G', \emptyset)$.

Pick a vertex $v \in W \setminus \mathsf{safe}_\alpha(\mathcal{D})$. Then $v \in \mathsf{Attr}_{\bar{\alpha}}(G, V_{\bar{\alpha}} \cap I)$, so player $\bar{\alpha}$ must have a strategy to force play to $V_{\bar{\alpha}} \cap I$. Let $\sigma_{\bar{\alpha}}$ be this strategy. We define a new strategy $\sigma'_{\bar{\alpha}}$ which, for sequences $v_0 \ldots v_n$ for which $v_n \notin V_{\bar{\alpha}} \cap I$ is defined as $\sigma_{\bar{\alpha}}$, and for $v_n \in V_{\bar{\alpha}} \cap I$, we have $\sigma'_{\bar{\alpha}}(v_0 \ldots v_n) = z$. Let σ_α be an arbitrary strategy for player α. Consider a play π, starting in v, which is consistent with both σ_α and $\sigma'_{\bar{\alpha}}$. Then there

must be a vertex w on that play such that $w \in I$, since $\sigma'_{\bar{\alpha}}$ forces play to $V_{\bar{\alpha}} \cap I$. But then, by construction, π must be finite and end in sink $z \notin W$. Note that vertex $z \in V'$ is won by $\bar{\alpha}$, so we find that π is won by $\bar{\alpha}$. Since σ_α is an arbitrary strategy by player α, we can conclude that all plays consistent with $\sigma'_{\bar{\alpha}}$ are won by $\bar{\alpha}$ and therefore vertex v is won by $\bar{\alpha}$, too. Hence, all vertices in $W \setminus \mathsf{safe}_\alpha(\mathcal{D})$ are won by $\bar{\alpha}$ in $\mathcal{D}'$. □

As a corollary of the above lemma, we find that α-dominions that contain vertices outside of the α-safe set are not guaranteed to be dominions in all extensions of the incomplete parity game.

Corollary 4.2.11. Let $\mathcal{D}$ be an incomplete parity game. Suppose that W is an α-dominion in $\mathcal{D}$. If $W \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$, then there is an (incomplete) parity game $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$ and W is *not* an α-dominion in $\mathcal{D}'$.

The theorem below summarises the two previous results, claiming that the sets $\mathsf{safe}_\Diamond(\mathcal{D})$ and $\mathsf{safe}_\Box(\mathcal{D})$ are the optimal subsets that can be used safely when combining solving and the exploration of a parity game.

Theorem 4.2.12. Let $\mathcal{D} = (G, I)$, with $G = (V, E, p, (V_\Diamond, V_\Box))$, be an incomplete parity game. Define W_α as the union of all α-dominions in $\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$, and let $W_? = V \setminus (W_\Diamond \cup W_\Box)$. Then $W_?$ is the largest set of vertices v for which there are incomplete parity games $\mathcal{D}^\alpha$ and $\mathcal{D}^{\bar{\alpha}}$ such that $\mathcal{D} \sqsubseteq \mathcal{D}^\alpha$ and $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ and v is won by α in $\mathcal{D}^\alpha$ and v is won by $\bar{\alpha}$ in $\mathcal{D}^{\bar{\alpha}}$.

Proof. Let $\mathcal{D}$, with $G = (V, E, p, (V_\Diamond, V_\Box))$ be an incomplete parity game. Pick a vertex $v \in W_?$. Suppose that in G, vertex $v \in W_?$ is won by player α. Let $\mathcal{D}^\alpha = \mathcal{D}$. Then $\mathcal{D} \sqsubseteq \mathcal{D}^\alpha$ and v is also won by α in $\mathcal{D}^\alpha$.

Next, we argue that there must be a game $\mathcal{D}^{\bar{\alpha}}$ such that $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ and v is won by $\bar{\alpha}$ in $\mathcal{D}^{\bar{\alpha}}$. Since $v \in W_?$ is won by player α in G, v must belong to an α-dominion in G. Towards a contradiction, assume that $v \in \mathsf{safe}_\alpha(\mathcal{D})$. Then there must also be a α-dominion containing v in $G \cap \mathsf{safe}_\alpha(\mathcal{D})$, since $\bar{\alpha}$ cannot escape the set $\mathsf{safe}_\alpha(\mathcal{D})$. But then $v \in W_\alpha$. Contradiction, so $v \notin \mathsf{safe}_\alpha(\mathcal{D})$. So, v must be part of an α-dominion D in G such that $D \not\subseteq \mathsf{safe}_\alpha(\mathcal{D})$. By Lemma 4.2.10, we find that there is an incomplete parity game $\mathcal{D}^{\bar{\alpha}}$ such that $\mathcal{D} \sqsubseteq \mathcal{D}^{\bar{\alpha}}$ and all vertices in $D \setminus \mathsf{safe}_\alpha(\mathcal{D})$, and vertex $v \in D$ in particular, are won by $\bar{\alpha}$ in $\mathcal{D}^{\bar{\alpha}}$.

Finally, we argue that $W_?$ cannot be larger. Pick a vertex $v \notin W_?$. Then there must be some player α such that $v \in W_\alpha$, and, consequently, there must be an α-dominion $D \subseteq \mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D})$ such that $v \in D$. But then by Lemma 4.2.8, we find that v is won by α in all incomplete parity games $\mathcal{D}'$ such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. □

4.3 On-the-fly Solving

In the previous section we saw that for any solver solve_α, which accepts a parity game as input and returns an α-dominion W_α, a correct on-the-fly solving algorithm can be obtained by computing $W_\alpha = \mathsf{solve}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}))$ while exploring an (incomplete) parity game $\mathcal{D}$. While this approach is clearly sound, computing the set of safe vertices can be expensive for large state spaces and potentially wasteful when no dominions are found afterwards. In the next section, we introduce *safe attractors* which, as we show, can be used to search for specific dominions without first computing the α-safe set of vertices. A similar notion was also introduced in [21].

4.3.1 Safe Attractors

We start by observing that the α-attractor to a set U in an incomplete parity game $\mathcal{D}$ does not make a distinction between the set of complete and incomplete vertices. Consequently, it may wrongly conclude that α has a strategy to force the play to U when the attractor strategy involves incomplete vertices owned by $\bar{\alpha}$. We thus need to make sure that such vertices are excluded from consideration. This can be achieved by considering the set of *unsafe* vertices $V_{\bar{\alpha}} \cap I$ as potential vertices that can be used by the other player to escape. We define the safe α-attractor as the least fixed point of the *safe* control predecessor. The latter is defined as follows for a set of vertices U:

$$\mathsf{spre}_\alpha(\mathcal{D}, U) = (V_\alpha \cap \mathsf{pre}(\mathcal{D}, U)) \cup (V_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathcal{D}, V \setminus U) \cup \mathsf{sinks}(\mathcal{D}) \cup I))$$

Lemma 4.3.1. Let $\mathcal{D}$ be an incomplete parity game. For all vertex sets $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ it holds that $\mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), X) = \mathsf{spre}_\alpha(\mathcal{D}, X)$.

Proof. Let $\mathcal{D} = (G, I)$, with $G = (V, E, p, (V_\Diamond, V_\Box))$, be an incomplete parity game and let $\mathcal{D}' = \mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}) = (G', I \cap \mathsf{safe}_\alpha(\mathcal{D}))$, with $G' = (V', E', p', (V'_\Diamond, V'_\Box))$. Pick an arbitrary vertex set $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$.

- *ad* $\mathsf{cpre}_\alpha(\mathcal{D}', X) \subseteq \mathsf{spre}_\alpha(\mathcal{D}, X)$. Let $v \in \mathsf{cpre}_\alpha(\mathcal{D}', X) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. We distinguish two cases: $v \in V'_\alpha$ and $v \in V'_{\bar{\alpha}}$.
  - Case $v \in V'_\alpha$. Then, by definition of the control predecessor, also $v \in V'_\alpha \cap \mathsf{pre}(\mathcal{D}', X)$. Since $V'_\alpha \subseteq V_\alpha$, it suffices to show that $v \in \mathsf{pre}(\mathcal{D}, X)$. But this follows instantly since $\mathcal{D}'$ is a substructure of $\mathcal{D}$. Hence, $v \in V_\alpha \cap \mathsf{pre}(\mathcal{D}', X) \subseteq \mathsf{spre}_\alpha(\mathcal{D}, X)$.
  - Case $v \in V'_{\bar{\alpha}}$. Since $v \in \mathsf{cpre}_\alpha(\mathcal{D}', X)$, we then also have $v \in V'_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathcal{D}', V' \setminus X) \cup \mathsf{sinks}(\mathcal{D}'))$, so $v \notin \mathsf{pre}(\mathcal{D}', V' \setminus X) \cup \mathsf{sinks}(\mathcal{D}')$.

Suppose $v \in I$. Since $v \in V'_{\bar{\alpha}} \subseteq V_{\bar{\alpha}}$, also $v \in V_{\bar{\alpha}} \cap I$, and therefore $v \in \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D}, V_{\bar{\alpha}} \cap I)$. But this contradicts $v \in \mathsf{safe}_\alpha(\mathcal{D}) = V \setminus \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D}, V_{\bar{\alpha}} \cap I)$. Hence, $v \notin I$. Next, suppose that $v \in \mathsf{sinks}(\mathcal{D})$. Since $v \notin \mathsf{sinks}(\mathcal{D}')$, vertex v was removed from $\mathcal{D}'$, and, hence, $v \notin \mathsf{safe}_\alpha(\mathcal{D})$. Contradiction, so $v \notin \mathsf{sinks}(\mathcal{D})$. Finally, suppose that $v \in \mathsf{pre}(\mathcal{D}, V \setminus X)$. Then there must be a vertex $w \in vE$ for which $w \in V \setminus X$ and $w \notin V' \setminus X$. However, then $w \notin \mathsf{safe}_\alpha(\mathcal{D})$ and also $v \notin \mathsf{safe}_\alpha(\mathcal{D})$ since $v \in V_{\bar{\alpha}}$. So, $v \notin \mathsf{pre}(\mathcal{D}, V \setminus X)$. Hence, we can conclude that $v \in V_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathcal{D}, V \setminus X) \cup \mathsf{sinks}(\mathcal{D}) \cup I)$.

We may thus conclude that $v \in \mathsf{spre}_\alpha(\mathcal{D}, X)$.

- *ad* $\mathsf{cpre}_\alpha(\mathcal{D}', X) \supseteq \mathsf{spre}_\alpha(\mathcal{D}, X)$. Let $v \in \mathsf{spre}_\alpha(\mathcal{D}, X)$. We again use a case distinction to show that $v \in \mathsf{cpre}_\alpha(\mathcal{D}', X)$.
  - Case $v \in V_\alpha$. Then also $v \in \mathsf{pre}(\mathcal{D}, X)$, so for some $w \in vE$, we have $w \in X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. But then also $v \in \mathsf{safe}_\alpha(\mathcal{D})$, since α can move to a vertex in X, preventing plays passing through v from reaching $V_{\bar{\alpha}} \cap I$. Hence, $v \in V_\alpha \cap \mathsf{safe}_\alpha(\mathcal{D}) = V'_\alpha$. Since $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$, we also have $v \in \mathsf{pre}(\mathcal{D}', X)$. Hence, $v \in V'_\alpha \cap \mathsf{pre}(\mathcal{D}', X)$.
  - Case $v \in V_{\bar{\alpha}}$. Then $v \notin \mathsf{pre}(\mathcal{D}, V \setminus X) \cup \mathsf{sinks}(\mathcal{D}) \cup I$. Assume that $v \in \mathsf{pre}(\mathcal{D}', V' \setminus X)$. Since $\mathcal{D}'$ is a subgraph of $\mathcal{D}$ and $V' \setminus X \subseteq V \setminus X$, we then have $v \in \mathsf{pre}(\mathcal{D}, V \setminus X)$. Contradiction. Next, assume that $v \in \mathsf{sinks}(\mathcal{D}')$. But this contradicts $v \notin \mathsf{sinks}(\mathcal{D})$, since $\mathsf{sinks}(\mathcal{D}') \subseteq \mathsf{sinks}(\mathcal{D})$. Finally, we know that $v \notin \mathsf{pre}(\mathcal{D}, V \setminus X)$. This means that all successors of v must be in X, *i.e.*, we find that $vE \subseteq X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. But then, since $v \notin \mathsf{sinks}(\mathcal{D}) \cup I$, we have $v \notin \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D}, V_{\bar{\alpha}} \cap I)$, so $v \in \mathsf{safe}_\alpha(\mathcal{D})$. Hence, $v \in V'_\alpha$.

  We may therefore conclude that $v \in \mathsf{cpre}_\alpha(\mathcal{D}', X)$. □

The safe α-attractor to U, denoted $\mathsf{SAttr}_\alpha(\mathcal{D}, U)$, is the set of vertices from which player α can force to *safely* reach U in $\mathcal{D}$:

$$\mathsf{SAttr}_\alpha(\mathcal{D}, U) = \mu Z.(U \cup \mathsf{spre}_\alpha(\mathcal{D}, Z))$$

Lemma 4.3.2. Let $\mathcal{D}$ be an incomplete parity game, and $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Then $\mathsf{Attr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), X) = \mathsf{SAttr}_\alpha(\mathcal{D}, X)$.

Proof. We show by means of an induction that the fixed point approximants A_i of $\mathsf{Attr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), X)$ are equal to the approximants B_i of $\mathsf{SAttr}_\alpha(\mathcal{D}, X)$ and that $A_i \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Initially, $A_0 = B_0 = \emptyset$, which are equal and $\emptyset \subseteq \mathsf{safe}_\alpha(\mathcal{D})$.

Inductive step, assume that $A_i = B_i$ and $A_i \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. We can show that $X \cup \mathsf{cpre}_\alpha(G \cap \mathsf{safe}_\alpha(\mathcal{D}), A_i) = X \cup \mathsf{spre}_\alpha(\mathcal{D}, B_i)$. First, using Lemma 4.3.1 we can conclude that $\mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), A_i) = \mathsf{spre}_\alpha(\mathcal{D}, A_i)$. From $A_i = B_i$ it follows that

$X \cup \mathsf{cpre}_\alpha(G \cap \mathsf{safe}_\alpha(\mathcal{D}), A_i) = X \cup \mathsf{spre}_\alpha(\mathcal{D}, B_i)$. Finally, it holds that $\mathsf{cpre}_\alpha(G \cap \mathsf{safe}_\alpha(\mathcal{D}), A_i) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ by definition. □

In particular, we can conclude the following:

Corollary 4.3.3. Let $\mathcal{D}$ be an incomplete parity game, and $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ be an α-dominion. Then $\mathsf{SAttr}_\alpha(\mathcal{D}, X)$ is an α-dominion for all $\mathcal{D}'$ satisfying $\mathcal{D} \sqsubseteq \mathcal{D}'$.

One application of the above corollary is the following: since on-the-fly solving is typically performed repeatedly, previously found dominions can be expanded by computing the safe α-attractor towards these already solved vertices.

Another corollary is the following, which states that we can always safely attract towards complete sinks.

Corollary 4.3.4. Let $\mathcal{D} = (G, I)$ be an incomplete parity game and let $\mathcal{D}'$ be such that $\mathcal{D} \sqsubseteq \mathcal{D}'$. Then $\mathsf{SAttr}_\alpha(\mathcal{D}, \mathsf{sinks}_{\bar{\alpha}}(\mathcal{D}) \setminus I)$ is an α-dominion in $\mathcal{D}'$.

4.3.2 Partial Solvers

In practice, a full-fledged solver, such as Zielonka's algorithm [140] or one of the Priority Promotion variants [4], may be costly to run often while exploring a parity game. Instead, cheaper partial solvers may be used that search for a dominion of a particular shape. We study three such partial solvers in this section, with a particular focus on solvers that lend themselves for parity games that are represented symbolically using, *e.g.*, BDDs [22], MDDs [102] or LDDs [41]. For the remainder of this section, we fix an arbitrary incomplete parity game $\mathcal{D} = ((V, E, p, (V_\Diamond, V_\Box)), I)$.

Winning solitaire cycles. A simple cycle in $\mathcal{D}$ can be represented by a finite sequence of distinct vertices $v_0 v_1 \ldots v_n$ satisfying $v_0 \in v_n E$. Such a cycle is an *α-solitaire cycle* whenever all vertices on that cycle are owned by player α.

Observe that if all vertices on an α-solitaire cycle have a priority that is of the same parity as the owner α, then all vertices on that cycle are won by player α. Formally, these are thus cycles through vertices in the set $P_\alpha \cap V_\alpha$, where $P_\Diamond = \{v \in V \setminus \mathsf{sinks}(\mathcal{D}) \mid p(v) \mod 2 = 0\}$ and $P_\Box = \{v \in V \setminus \mathsf{sinks}(\mathcal{D}) \mid p(v) \mod 2 = 1\}$. Let $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D})$ represent the largest set of α-solitaire winning cycles. Then $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) = \nu Z.(P_\alpha \cap V_\alpha \cap \mathsf{pre}(\mathcal{D}, Z))$.

Proposition 4.3.5. The set $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D})$ is an α-dominion and we have $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$.

Proof. We first prove that $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. We show, by means of an induction on the fixed point approximants A_i of the attractor, that $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) \cap \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D}, V_{\bar{\alpha}} \cap I) = \emptyset$. The base case follows immediately, as $\mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) \cap A_0 = \mathscr{C}^\alpha_{\mathsf{sol}}(\mathcal{D}) \cap \emptyset = \emptyset$. For the

induction, we assume that $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset) \cap A_i = \emptyset$; we show that also $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset) \cap ((V_{\bar{\alpha}} \cap I) \cup \mathsf{cpre}_{\bar{\alpha}}(\supset, A_i)) = \emptyset$. First, observe that $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset) \subseteq V_\alpha$; hence, it suffices to prove that $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset) \cap (V_\alpha \setminus (\mathsf{pre}(\supset, V \setminus A_i) \cup \mathsf{sinks}(\supset))) = \emptyset$. But this follows immediately from the fact that for every vertex $v \in \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$, we have $v \in P_\alpha \cap V_\alpha \cap \mathsf{pre}(\supset, \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset))$; more specifically, we have $vE \cap \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset) \neq \emptyset$ for all $v \in \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$.

The fact that $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$ is an α-dominion follows from the fact that for every vertex $v \in \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$, there is some $w \in vE \cap \mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$. This means that player α must have a strategy that is closed on $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$. Since all vertices in $\mathscr{C}^{\alpha}_{\mathsf{sol}}(\supset)$ are of the priority that is beneficial to α, this closed strategy is also winning for α. □

Observe that winning solitaire cycles can be computed without first computing the α-safe set. Parity games that stand to profit from detecting winning solitaire cycles are those originating from verifying safety properties.

Winning forced cycles. In general, a cycle in $\mathsf{safe}_\alpha(\supset)$, through vertices in $P_\Diamond$ can contain vertices of both players, providing player □ an opportunity to break the cycle if that is beneficial to her. Nevertheless, if breaking a cycle always inadvertently leads to another cycle through $P_\Diamond$, then we may conclude that all vertices on these cycles are won by player ♢. We call these cycles *winning forced cycles* for player ♢. A dual argument applies to cycles through $P_\Box$. Let $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$ represent the largest set of vertices that are on winning forced cycles for player α. More formally, we define $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) = \nu Z.(P_\alpha \cap \mathsf{safe}_\alpha(\supset) \cap \mathsf{cpre}_\alpha(\supset, Z))$.

Lemma 4.3.6. The set $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$ is an α-dominion and we have $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) \subseteq \mathsf{safe}_\alpha(\supset)$.

Proof. The fact that $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) \subseteq \mathsf{safe}_\alpha(\supset)$ follows immediately from the fact that for all $v \in \mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$, we have $v \in P_\alpha \cap \mathsf{safe}_\alpha(\supset) \cap \mathsf{cpre}_\alpha(\supset, \mathscr{C}^{\alpha}_{\mathsf{for}}(\supset))$.

We next show that $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$ is an α-dominion. Pick an arbitrary vertex $v \in \mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$. If $v \in V_\alpha$, then $vE \cap \mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) \neq \emptyset$, so player α has a strategy to move to another vertex in $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$. In case $v \in V_{\bar{\alpha}}$, then $vE \subseteq \mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$, so any play passing through v is guaranteed to next visit a vertex in $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$. Hence, player α has a closed strategy on $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$. Since $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) \subseteq P_\alpha$, such a closed strategy must be winning for α. □

A possible downside of the above construction is that it again requires to first compute $\mathsf{safe}_\alpha(\supset)$, which, in particular cases, may incur an additional overhead. Instead, we can compute the same set using the safe control predecessor. We define $\mathscr{C}^{\alpha}_{\mathsf{s\text{-}for}}(\supset) = \nu Z.(P_\alpha \cap \mathsf{spre}_\alpha(\supset, Z))$.

Proposition 4.3.7. We have $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset) = \mathscr{C}^{\alpha}_{\mathsf{s\text{-}for}}(\supset)$.

Proof. Let $\tau(Z) = P_\alpha \cap \mathsf{spre}_\alpha(\supset, Z)$. We use set inclusion to show that $\mathscr{C}^{\alpha}_{\mathsf{for}}(\supset)$ is indeed a fixed point of τ.

- *ad* $\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}) \subseteq \tau(\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. Pick a vertex $v \in \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})$. By definition of $\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})$, we have $v \in P_\alpha \cap \mathsf{safe}_\alpha(\mathcal{D}) \cap \mathsf{cpre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. We can observe that $\mathsf{safe}_\alpha(\mathcal{D}) \cap \mathsf{cpre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) = \mathsf{safe}_\alpha(\mathcal{D}) \cap \mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. But then, since $\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$, we find, by Lemma 4.3.1, that $\mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) = \mathsf{spre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. Hence, $v \in P_\alpha \cap \mathsf{spre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) = \tau(\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$.
- *ad* $\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}) \supseteq \tau(\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. Again pick a vertex $v \in \tau(\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. Then $v \in P_\alpha \cap \mathsf{spre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. Since $\mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$, by Lemma 4.3.1, we again have $\mathsf{spre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) = \mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. But then it must be the case that $v \in \mathsf{safe}_\alpha(\mathcal{D})$. Moreover, $\mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}), \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) \subseteq \mathsf{cpre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D}))$. So $v \in P_\alpha \cap \mathsf{safe}_\alpha(\mathcal{D}) \cap \mathsf{cpre}_\alpha(\mathcal{D}, \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})) = \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})$.

We show next that for any $Z = \tau(Z)$, we have $Z \subseteq \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})$. Let Z be such. We first show that for every $v \in Z \cap V_\alpha$, there is some $w \in vE \cap Z$, and for every $v \in Z \cap V_{\bar{\alpha}}$, we have $v \notin \mathsf{sinks}(\mathcal{D})$, $v \notin I$ and $vE \subseteq Z$. Pick $v \in Z \cap V_\alpha$. Then $v \in \tau(Z) \cap V_\alpha = P_\alpha \cap V_\alpha \cap \mathsf{spre}_\alpha(\mathcal{D}, Z) \subseteq \mathsf{pre}(\mathcal{D}, Z)$. But then $vE \cap Z \neq \emptyset$. Next, let $v \in Z \cap V_{\bar{\alpha}}$. Then $v \in \tau(Z) \cap V_{\bar{\alpha}} = P_\alpha \cap V_{\bar{\alpha}} \cap \mathsf{spre}_\alpha(\mathcal{D}, Z) \subseteq V_{\bar{\alpha}} \setminus (\mathsf{pre}(\mathcal{D}, V \setminus Z) \cup \mathsf{sinks}(\mathcal{D}) \cup I)$. So $v \notin \mathsf{pre}(\mathcal{D}, V \setminus Z) \cup \mathsf{sinks}(\mathcal{D}) \cup I$. Consequently, $vE \subseteq Z$, $v \notin \mathsf{sinks}(\mathcal{D})$ and $v \notin I$.

Since for every $v \in Z \cap V_\alpha$, we have $vE \cap Z \neq \emptyset$, there must be a strategy for player α to move to another vertex in Z. Let σ be this strategy. Moreover, since for all $v \in Z \cap V_{\bar{\alpha}}$ we have $vE \subseteq Z$, we find that σ is closed on Z and since $Z \cap \mathsf{sinks}(\mathcal{D}) = \emptyset$, strategy σ induces forced cycles. Moreover, since $Z \subseteq P_\alpha$, we can conclude that all vertices in Z are on winning forced cycles.

Finally, we must argue that $Z \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. But this follows from the fact that $Z \cap V_{\bar{\alpha}} \cap I = \emptyset$, and, hence, also $Z \cap \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D}, V_{\bar{\alpha}} \cap I) = \emptyset$. Since Z is contained within $P_\alpha \cap \mathsf{safe}_\alpha(\mathcal{D})$, we find that $Z \subseteq \mathscr{C}^{\alpha}_{\mathsf{for}}(\mathcal{D})$. □

Fatal attractors. Both solitaire cycles and forced cycles utilise the fact that the parity winning condition becomes trivial if the only priorities that occur on a play are of the parity of a single player. Fatal attractors [81] were originally conceived to solve parts of a game using algorithms that have an appealing worst-case running time; for a detailed account, we refer to [81]. While *ibid.* investigates several variants, the main idea behind a fatal attractor is that it identifies cycles in which the priorities are non-decreasing until the dominating priority of the attractor is (re)visited. We focus on a simplified (and cheaper) variant of the `psolB` algorithm of [81], which is based on the concept of a *monotone* attractor, which, in turn, relies on the monotone control predecessor defined below, where $P^{\geq c} = \{v \in V \mid p(v) \geq c\}$:

$$\mathsf{Mcpre}_\alpha(\mathcal{D}, Z, U, c) = P^{\geq c} \cap \mathsf{cpre}_\alpha(\mathcal{D}, Z \cup U)$$

The monotone attractor for a given priority is then defined as the least fixed point of the monotone control predecessor for that priority, formally $\mathsf{MAttr}_\alpha(\mathcal{D}, U, c) =$

$\mu Z.\mathsf{Mcpre}_\alpha(\mathcal{D},Z,U,c)$. A *fatal* attractor for priority c is then the largest set of vertices closed under the monotone attractor for priority c; *i.e.*, $\mathscr{F}^\alpha(\mathcal{D},c) = \nu Z.(P^{=c} \cap \mathsf{safe}_\alpha(\mathcal{D}) \cap \mathsf{MAttr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),Z,c))$, where $P^{=c} = P^{\geq c} \setminus P^{\geq c+1}$.

Lemma 4.3.8 (See [81], Theorem 2)**.** For even priority c, we have that $\mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),\mathscr{F}^\Diamond(\mathcal{D},c),c) \subseteq \mathsf{safe}_\Diamond(\mathcal{D})$ and also $\mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),\mathscr{F}^\Diamond(\mathcal{D},c),c)$ is an $\Diamond$-dominion. If c is odd then we have $\mathsf{MAttr}_\Box(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),\mathscr{F}^\Box(\mathcal{D},c),c) \subseteq \mathsf{safe}_\Box(\mathcal{D})$ and $\mathsf{MAttr}_\Box(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),\mathscr{F}^\Box(\mathcal{D},c),c)$ is an $\Box$-dominion.

Our simplified version of the `psolB` algorithm, here dubbed $\texttt{solB}^-$ computes fatal attractors for all priorities in descending order, accumulating $\Diamond$ and $\Box$-dominions and extending these dominions using a standard $\Diamond$ or $\Box$-attractor. This can be implemented using a simple loop over these priorities.

In line with the previous solvers, we can also modify this solver to employ a safe monotone control predecessor, which uses a construction that is similar in spirit to that of the safe control predecessor. Formally, we define the safe monotone control predecessor as follows:

$$\mathsf{sMcpre}_\alpha(\mathcal{D},Z,U,c) = P^{\geq c} \cap \mathsf{spre}_\alpha(\mathcal{D},Z \cup U)$$

The corresponding safe monotone α-attractor, denoted $\mathsf{sMAttr}_\alpha(\mathcal{D},U,c)$, is defined as follows: $\mathsf{sMAttr}_\alpha(\mathcal{D},U,c) = \mu Z.\mathsf{sMcpre}_\alpha(\mathcal{D},Z,U,c)$. We define the *safe* fatal attractor for priority c as the set $\mathscr{F}_\mathsf{s}^\alpha(\mathcal{D},c) = \nu Z.(P^{=c} \cap \mathsf{sMAttr}_\alpha(\mathcal{D},Z,c))$. Similar to the safe attractor case using a standard inductive argument we can show the following.

Lemma 4.3.9. Let $\mathcal{D}$ be an incomplete parity game, and $X \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Then it holds that $\mathsf{MAttr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),X,c) = \mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$ for any player α and priority c.

Proof. We show by means of an induction that the fixed point approximants A_i of $\mathsf{MAttr}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),X,c)$ are equal to the approximants B_i of $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$ and $A_i \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Initially, $A_0 = B_0 = \emptyset$ and $\emptyset \subseteq \mathsf{safe}_\alpha(\mathcal{D})$.

Inductive step, assume that $A_i = B_i$ and $A_i \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. We observe that $\mathsf{Mcpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),A_i,X,c) = P^{\geq c} \cap \mathsf{cpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),X \cup A_i)$, which by Lemma 4.3.1 is equal to $P^{\geq c} \cap \mathsf{spre}_\alpha(\mathcal{D},X \cup A_i)$ since $X \cup A_i \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. Therefore, $\mathsf{Mcpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),A_i,X,c) = \mathsf{sMcpre}_\alpha(\mathcal{D},A_i,X,c)$. Furthermore, it also holds that $\mathsf{Mcpre}_\alpha(\mathcal{D} \cap \mathsf{safe}_\alpha(\mathcal{D}),A_i,X,c) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$. □

Lemma 4.3.10. Let $\mathcal{D}$ be an incomplete parity game and X a set of vertices such that $X \subseteq \mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$. Then $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \subseteq \mathsf{safe}_\alpha(\mathcal{D})$ for any player α and priority c.

Proof. Let $\mathcal{D} = (G,I)$, with $G = (V,E,p,(V_\Diamond,V_\Box))$, be an incomplete parity game. We show, by means of an induction on the fixed point approximants A_i of the attractor,

that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap \mathsf{Attr}_{\bar{\alpha}}(\mathcal{D},V_{\bar{\alpha}} \cap I) = \emptyset$. The base case follows immediately, as $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap A_0 = \mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap \emptyset = \emptyset$.

For the induction, we assume that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap A_i = \emptyset$; we show that also $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap ((V_{\bar{\alpha}} \cap I) \cup \mathsf{cpre}_{\bar{\alpha}}(\mathcal{D},A_i)) = \emptyset$. First of all, since $\mathsf{spre}_\alpha(\mathcal{D},Z) \cap V_{\bar{\alpha}} \cap I = \emptyset$, for any Z, it follows that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap V_{\bar{\alpha}} \cap I = \emptyset$. It remains to show that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap \mathsf{cpre}_{\bar{\alpha}}(\mathcal{D},A_i) = \emptyset$, which is equal to showing that $P^{\geq c} \cap \mathsf{spre}_\alpha(\mathcal{D},\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cup X) \cap \mathsf{cpre}_{\bar{\alpha}}(\mathcal{D},A_i) = \emptyset$. Since $X \subseteq \mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$ we have that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cup X = \mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$, and therefore it suffices to show that $\mathsf{spre}_\alpha(\mathcal{D},\mathsf{sMAttr}_\alpha(\mathcal{D},X,c)) \cap \mathsf{cpre}_{\bar{\alpha}}(\mathcal{D},A_i) = \emptyset$. Consider any vertex v in the set $\mathsf{spre}_\alpha(\mathcal{D},\mathsf{sMAttr}_\alpha(\mathcal{D},X,c))$. We distinguish two cases. If $v \in V_\alpha$ then we have that $vE \cap \mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \neq \emptyset$ and thus $v \notin (V_\alpha \setminus (\mathsf{pre}(\mathcal{D},V \setminus A_i) \cup \mathsf{sinks}(\mathcal{D})))$. Otherwise, $v \in V_{\bar{\alpha}}$ and we have that $vE \subseteq \mathsf{sMAttr}_\alpha(\mathcal{D},X,c)$ and therefore $v \notin V_{\bar{\alpha}} \cap \mathsf{pre}(\mathcal{D},A_i)$.

Thus we conclude that $\mathsf{sMAttr}_\alpha(\mathcal{D},X,c) \cap \mathsf{cpre}_{\bar{\alpha}}(\mathcal{D},A_i) = \emptyset$. □

We conclude with the correctness of safe fatal attractors.

Proposition 4.3.11. Let $\mathcal{D}$ be an incomplete parity game. We have $\mathscr{F}_\mathsf{s}^\Diamond(\mathcal{D},c) = \mathscr{F}^\Diamond(\mathcal{D},c)$ for even c and for odd c we have $\mathscr{F}_\mathsf{s}^\Box(\mathcal{D},c) = \mathscr{F}^\Box(\mathcal{D},c)$.

Proof. The proof proceeds along the lines of that of Proposition 4.3.7. Let c be an even priority and let $\tau(Z) = P^{=c} \cap \mathsf{sMAttr}_\Diamond(\mathcal{D},Z,c)$. We show that $\mathscr{F}^\Diamond(\mathcal{D},c)$ is indeed a fixed point of τ.

By definition of $\mathscr{F}^\Diamond(\mathcal{D},c)$, we know that $\mathscr{F}^\Diamond(\mathcal{D},c)$ is equal to $P^{=c} \cap \mathsf{safe}_\Diamond(\mathcal{D}) \cap \mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\Diamond(\mathcal{D}),\mathscr{F}^\Diamond(\mathcal{D},c),c)$. Since $\mathscr{F}^\Diamond(\mathcal{D},c) \subseteq \mathsf{safe}_\Diamond(\mathcal{D})$ we know that the monotone attractor $\mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\Diamond(\mathcal{D}),\mathscr{F}^\Diamond(\mathcal{D},c),c) = \mathsf{sMAttr}_\Diamond(\mathcal{D},\mathscr{F}^\Diamond(\mathcal{D},c),c)$ by Lemma 4.3.9. Furthermore, $\mathsf{sMAttr}_\Diamond(\mathcal{D},\mathscr{F}^\Diamond(\mathcal{D},c),c) \subseteq \mathsf{safe}_\Diamond(\mathcal{D})$ by Lemma 4.3.10 and thus $\mathscr{F}^\Diamond(\mathcal{D},c) = P^{=c} \cap \mathsf{sMAttr}_\Diamond(\mathcal{D},\mathscr{F}^\Diamond(\mathcal{D},c),c) = \tau(\mathscr{F}^\Diamond(\mathcal{D},c))$.

Next, we show that for any $Z = \tau(Z)$ it holds that $Z \subseteq \mathscr{F}^\Diamond(\mathcal{D},c)$. Let Z be such. By Lemma 4.3.10 we have that $Z \subseteq \mathsf{safe}_\Diamond(\mathcal{D})$ since $Z \subseteq \mathscr{F}_\mathsf{s}^\Diamond(\mathcal{D},c)$ and $\mathscr{F}_\mathsf{s}^\Diamond(\mathcal{D},c) \subseteq \mathsf{sMAttr}_\Diamond(\mathcal{D},\mathscr{F}_\mathsf{s}^\Diamond(\mathcal{D},c),c)$. Therefore, by Lemma 4.3.9 it follows that $\mathsf{sMAttr}_\Diamond(\mathcal{D},Z,c) = \mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\Diamond(\mathcal{D}),Z,c)$. Thus we conclude that $Z = P^{=c} \cap \mathsf{safe}_\Diamond(\mathcal{D}) \cap \mathsf{MAttr}_\Diamond(\mathcal{D} \cap \mathsf{safe}_\Diamond(\mathcal{D}),Z,c)$ and therefore $Z \subseteq \mathscr{F}^\Diamond(\mathcal{D},c)$.

The proof for odd priority c is completely dual. □

Similar to algorithm $\mathtt{solB}^-$, the algorithm $\mathtt{solB}_s^-$ computes safe fatal attractors for priorities in descending order and collects the safe-α-attractor extended dominions obtained this way.

4.4 Implementation

We have implemented a symbolic exploration technique for parity games in the mCRL2 toolset [23]. The toolset converts each model and property combination into a so-called parameterised Boolean equation system [70], abbreviated as PBES. The properties are written in the modal μ-calculus with data [69] and the models in the mCRL2 specification language [63].

We have implemented a translation from a PBES into a format that straightforwardly encodes the underlying parity game based on the translation presented in [89]. From this syntactic representation we derive a so-called symbolic parity game by means of an exploration algorithm. These symbolic parity games are parity games where the vertices V of the underlying graph are (n-)tuples of natural numbers. In these tuples we encode the player and priority of every vertex and the data parameters of the PBES. These data parameters are derived from the model and property and occur as parameters in the PBES. The natural numbers are mapped to complex data such as lists and sets that can be specified in the mCRL2 specification language.

The vertices and edges of the symbolic parity game can be efficiently encoded using multi-valued decision diagrams [102], abbreviated as MDD. In the implementation we use MDD-like data structures called *List Decision Diagrams (LDDs)*, and the corresponding Sylvan implementation [41], for this representation. Sylvan offers efficient implementations for set operations and relational operations such as successors and predecessors.

The effectiveness of our on-the-fly reachability analysis relies, to a large extent, on the assumption that the set of edges is the union of predominantly *sparse* relations that can be represented efficiently; *i.e.*, $E = E_1 \cup \cdots \cup E_m$. Our tool exploits techniques such as *read* and *write* dependencies [89, 12], and uses sophisticated exploration strategies such as *chaining* and *saturation* [29] for the exploration. We illustrate the basic idea of sparseness below.

Example 4.4.1. Consider the predicates $\mathscr{V}$ and $\mathscr{E}$ defined below, describing the graph (V,E), where x and y are variables and their primed counterparts represent the 'next state':

$$\begin{array}{ll} \mathscr{V}(x,y) & = x \leq 10 \wedge y \leq 2 \\ \mathscr{E}(x,y,x',y') & = (x' = (x+1) \bmod 10 \wedge y = y') \vee (x = x' \wedge y' = (y+1) \bmod 2) \end{array}$$

The set of edges E can be described as the union of set $E_1 = \{\langle (x,y),(x',y)\rangle \mid x' = (x+1) \bmod 10\}$ and $E_2 = \{\langle (x,y),(x,y')\rangle \mid y' = (y+1) \bmod 2\}$, restricted to V. Since the edges in E_1 only change the value of x, one can, intuitively, represent E_1 efficiently as a relation on shorter tuples: $E_1' = \{\langle x,x'\rangle \mid x' = (x+1) \bmod 10\}$; likewise, E_2 has a concise representation.

4.4.1 Solving Parity Games

For all three on-the-fly solving techniques of Section 4.3, we have implemented 1) a variant that runs the standard (partial) solver on the α-safe subgame and removes the found dominion using the standard attractor (within that subgame), and 2) a variant that uses (partial) solvers with the safe attractors. Moreover, we also conduct experiments using a full solver running on an α-safe subgame. This full solver is based on Zielonka's recursive algorithm [140], which remains one of the most competitive algorithms in practice, both explicitly and symbolically [124, 39].

An important design aspect is to decide how the exploration and the on-the-fly solving should interleave. For this we have implemented a time based heuristic that keeps track of the time spent on solving and exploration steps. The time measurements are used to ensure that (approximately) ten percent of total time is spent on solving by delaying the next call to the solver. We do not terminate the partial solver when it requires more time, and thus it is only approximate. As a result of this heuristic, cheap solvers will be called more frequently than more expensive (and more powerful) ones, which may cause the latter to explore larger parts of the game graph.

We have also considered two other heuristics that we have eventually discarded, but that could be revisited as future work. One simple heuristic is to use the number of breadth first search iterations as measure and perform solving every n iterations. The disadvantage here is that typically the number of vertices encountered at every level first grows exponentially, followed by a long tail where only relatively few new vertices are found. In this last section the solver would therefore be triggered often, but not much information is added. Therefore, another heuristic was based on the idea that in explicit representations it is often natural to consider the amount of vertices that have been explored as a measure. However, for symbolic representations there is not a direct correspondence between the amount of vertices represented and the number of nodes in the corresponding decision diagram. Therefore, a more natural approach was to consider the amount of nodes as a measure and perform solving when a number of new nodes had been added. In practice it turned out that computing the number of nodes can be expensive for large problems, which is undesirable for a heuristic. However, perhaps the implementation could be improved to avoid this overhead.

4.5 Experimental Results

We experimentally evaluate the techniques of Section 4.3. For this, we use games stemming from practical model checking and equivalence checking problems. Our experiments are run, single-threaded, on an Intel Xeon 6136 CPU @ 3 GHz PC. The sources for these experiments can be obtained from the downloadable artefact [92].

4.5.1 Cases

Table 4.1 provides an overview of the models and a description of the properties that are being checked. The properties are written in the modal μ-calculus with data [69]. For the equivalence checking case we have mutated the original model to introduce a defect. For each property, we indicate the *nesting depth* (ND) and *alternation depth* [30] and whether the parity game is *solitaire* (Yes/No). The nesting depth indicates how many different priorities occur in the resulting game; for our encoding this is at most ND+2 (the additional ones encode constants '*true*' and '*false*'). The alternation depth is an indication of a game's complexity due to alternating priorities.

Table 4.1: Models and formulas.

Model	Ref.	Prop.	Result	ND	AD	Sol.	Description
SWP	[132]	1	false	1	1	Y	No error transition
		2	false	3	3	N	Infinitely often enabled then infinitely often taken
WMS	[118]	1	false	1	1	Y	Job failed to be done
		2	false	1	1	Y	No zombie jobs
		3	true	3	2	Y	A job can become alive again infinitely often
		4	false	2	2	N	Branching bisimulation with a mutation
BKE	[11]	1	true	1	1	Y	No secret leaked
		2	false	2	1	N	No deadlock
CCP	[111]	1	false	2	1	N	No deadlock
		2	false	2	1	N	After access there is always accessover possible
PDI	n/a	1	true	2	1	N	Controller reaches state before it can connect again
		2	false	2	1	N	Connection impermissible can always happen or we establish a connection
		3	false	3	1	N	When connected move to not ready for connection and do not establish a connection until it is allowed again
		4	true	2	1	N	The interlocking moves to the state connection closed before it is allowed to succesfully establish a connection

We use MODEL-*i* to indicate the parity game belonging to model MODEL and property *i*. Models SWP, BKE and CCP are protocol specifications. The model PDI is a specification of a EULYNX SCI-LX SysML interface model that is used for a train interlocking system. Finally, WMS is the specification of a workload management system used at CERN. Using tools in mCRL2 [23], we have converted each model and property combination into a so-called parameterised Boolean equation systems [70], a higher-level logic that can be used to represent the underlying parity game. The toolset also contains tools to encode strong bisimilarity and related equivalences into a PBES based on the theory presented in [26].

Parity games SWP-1, WMS-1, WMS-2 and BKE-1 encode typical safety properties where some action should not be possible. In terms of the alternation-free modal mu-calculus with regular expressions, such properties are of the shape $[\mathsf{true}^*.\mathsf{a}]\mathsf{false}$. These properties are violated exactly when the vertex encoding '*false*' can be reached. Parity games SWP-2, WMS-3 and WMS-4 are more complex properties with alternating

priorities, where WMS-4 encodes branching bisimulation using the theory presented in [26]. The parity games BKE-2 and CCP-1 encode a 'no deadlock' property given by a formula which states that after every path there is at least one outgoing transition. Finally, CCP-2 and all PDI cases contain formulas with multiple fixed points that yield games with multiple priorities but no (dependent) alternation.

Table 4.2: Experiments with parity games where on-the-fly solving cannot terminate early. All run times are in seconds. The number of vertices is given in millions. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
BKE-1	full	40	640	65	**705**	14
	solitaire	40/40	629/615	153/100	782/715	15/15
	cycles	40/40	635/644	149/160	785/804	15/15
	fatal	40/40	624/625	152/164	776/789	15/15
	partial	40	651	147	798	15
PDI-1	full	114	27	0.1	**28**	2
	solitaire	114/114	28/27	4/0	33/**28**	2/2
	cycles	114/114	29/28	7/7	36/35	2/2
	fatal	114/114	28/28	4/7	32/35	2/2
	partial	114	28	9	37	2
PDI-4	full	474	286	0	**287**	2
	solitaire	474/474	284/281	46/14	331/295	2/2
	cycles	474/474	284/287	92/91	376/378	2/2
	fatal	474/474	285/283	80/91	365/374	2/2
	partial	474	286	64	350	2

4.5.2 Results

In Tables 4.2 and 4.3 we compare the on-the-fly solving strategies presented in Section 4.3. In the 'Strategy' column we indicate the on-the-fly solving strategy that is used. Here *full* refers to a complete exploration followed by solving with the Zielonka recursive algorithm. We use *solitaire* to refer to solitaire winning cycle detection, *cycles* for forced winning cycle detection, *fatal* to refer to fatal attractors and finally *partial* for on-the-fly solving with a Zielonka solver on safe regions. For solvers with a standard variant and a variant that utilises the safe attractors the first number in a cell in the table indicates the result of applying the (standard) solver on *safe* vertices, and the second number (following the slash '/') indicates the result when using the solver that utilises safe attractors.

The column 'Vertices' indicates the number of vertices explored in the game. In the next columns we indicate the time spent on exploring and solving specifically and the total time in seconds. We exclude the initialisation time that is common to all experiments. Finally, the last column indicates memory used by the tool in gigabytes. We report the average of 5 runs and have set a timeout (indicated by ‡) at 1200 seconds

Table 4.3: Experiments with parity games in which *at least one* partial solver terminates early. All run times are in seconds. The number of vertices is given in millions. For solvers with two variants the first number indicates the result of applying the solver on *safe* vertices, and following the slash '/' the result when using the solver that uses safe attractors. Memory is given in gigabytes. Bold-faced numbers indicate the lowest value.

Game	Strategy	Vertices (10^6)	Explore (s)	Solve (s)	Total (s)	Mem (GB)
SWP-1	full	13304	‡	n/a	‡	‡
	solitaire	15.1/0.4	8.5/1.4	27.3/0.1	35.8/**1.5**	2.8/1.5
	cycles	25.2/0.9	12.3/1.8	42.7/1.0	55.0/2.8	3.2/1.5
	fatal	15.1/0.4	9.0/1.3	29.4/0.4	38.4/1.7	3.1/1.5
	partial	27.1	13.1	50.4	63.5	3.6
SWP-2	full	1987	‡	n/a	‡	‡
	solitaire	1631/1987	‡/‡	163/11	‡/‡	‡/‡
	cycles	1774/1774	‡/‡	154/91	‡/‡	‡/‡
	fatal	0.007/0.007	0.9/0.9	0.4/0.2	1.3/**1.0**	1.4/1.2
	partial	0.007	0.9	0.4	1.3	1.4
WMS-1	full	270	2.8	0.4	3.3	0.2
	solitaire	270/240	2.8/2.5	0.8/0.4	3.6/**2.9**	0.3/0.2
	cycles	270/270	2.9/3.2	0.8/8.0	3.7/11.2	0.3/0.5
	fatal	270/270	2.6/3.2	0.8/8.5	3.4/11.7	0.3/0.5
	partial	270	2.7	0.8	3.5	0.3
WMS-2	full	317	3.3	0.3	3.6	0.2
	solitaire	7/7	0.2/0.2	1.0/0.5	1.2/**0.8**	0.1/0.1
	cycles	7/66	0.2/0.8	1.0/2.7	1.2/3.4	0.1/0.2
	fatal	7/66	0.2/0.7	1.0/2.9	1.3/3.6	0.1/0.2
	partial	7	0.2	1.1	1.3	0.1
WMS-3	full	317	2.6	0.1	2.7	0.2
	solitaire	317/317	2.6/2.6	0.4/0.3	3.1/2.9	0.2/0.2
	cycles	317/317	2.7/2.7	0.4/0.6	3.1/3.3	0.2/0.2
	fatal	5/1	0.2/0.1	0.5/0.1	0.7/**0.2**	0.1/0.1
	partial	5	0.2	0.3	0.5	0.1
WMS-4	full	366	‡	n/a	‡	‡
	solitaire	0.03/0.03	38/38	0.8/0.1	39/38	2/2
	cycles	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	fatal	0.03/0.03	37/37	0.8/0.3	38/**37**	2/2
	partial	0.03	37	0.7	38	2
BKE-2	full	119	942	36.5	979	28
	solitaire	0.0007/0.0001	0.2/0.1	0.0/0.0	**0.2/0.2**	0.9/0.9
	cycles	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2/0.2**	0.9/0.9
	fatal	0.0007/0.0003	0.2/0.2	0.0/0.0	**0.2/0.2**	0.9/0.9
	partial	0.0007	0.2	0.0	**0.2**	0.9
CCP-1	full	0.4	28	4.2	32	2
	solitaire	0.003/0.003	1.0/1.0	0.1/0.1	**1.1/1.1**	2/2
	cycles	0.003/0.003	1.0/1.0	0.1/0.1	**1.1/1.1**	2/2
	fatal	0.006/0.003	1.3/1.1	0.1/0.1	1.4/1.2	1.5/1.5
	partial	0.003	1.0	0.1	**1.1**	1.5
CCP-2	full	0.9	35	33	68	1.7
	solitaire	0.02/0.007	1.6/1.1	0.2/0.0	1.8/**1.1**	1.5/1.5
	cycles	0.02/0.007	1.9/1.1	0.2/0.1	2.1/1.2	1.5/1.5
	fatal	0.02/0.007	1.6/1.2	0.2/0.1	1.8/1.3	1.5/1.5
	partial	0.02	1.6	0.2	1.8	1.5
PDI-2	full	229	31	12	43	2
	solitaire	229/229	33/32	34/12	67/45	2/2
	cycles	30/30	15/14	3/5	**17**/19	2/2
	fatal	30/30	15/15	3/5	18/19	2/2
	partial	123	23	29	51	2
PDI-3	full	436	228	8	236	2
	solitaire	436/436	230/228	36/32	266/260	2/2
	cycles	78/162	65/102	19/64	84/166	2/2
	fatal	75/84	64/67	19/23	**83**/90	2/2
	partial	110	82	30	112	2

per run. Table 4.2 contains all benchmarks that require a full exploration of the game graph, providing an indication of the overhead in cases where this is unavoidable; Table 4.3 contains all benchmarks where *at least one* of the partial solvers allows exploration to terminate early.

For games SWP-1, WMS-1, WMS-2 in Table 4.3 we find that *solitaire*, and in particular the safe attractor variant, is able to determine the solution the fastest. Also, for all entries in Table 4.2 this is the solver with the least overhead. Next, we observe that for cases such as WMS-1 and PDI-3 using the safe attractor variants of the solvers can be detrimental. Our observation is that first computing safe sets (especially using chaining) can be quick when most vertices are owned by one player and one priority. On the other hand, the computation of the safe attractor requires computing the vertices of the opposing player that are forced to reach the target set in one step, which is more involved than computing the α-predecessors. There are also cases WMS-3, WMS-4, CCP-1 and CCP-2 where the safe attractor variants are faster and these cases all have multiple priorities. In cases where these solvers are slow (for example PDI-3) we also observe that more states are explored before termination, because the earlier mentioned time based heuristic results in calling the solver significantly less frequently.

For parity games SWP-2 and WMS-3 only *fatal* and *partial* are able to find a solution early, which shows that more powerful partial solvers can be useful. From Table 4.2 and the cases in which the safe attractor variants perform poorly we learn that the partial solvers can, as expected, cause overhead. This overhead is in our benchmarks on average 30 percent, but when it terminates early it can be very beneficial, achieving speed-ups of up to several orders of magnitude.

4.6 Conclusion

In this work we have developed the theory to reason about on-the-fly solving of parity games, independent of the strategy that is used to explore games. In addition to that, we have introduced the notion of *safe* vertices, shown their correctness, proven an optimality result, and we have studied partial solvers and shown that these can be made to run without determining the safe vertices first; which can be useful for on-the-fly solving. Finally, we have demonstrated the practical purpose of our method and observed that solitaire winning cycle detection with safe attractors is almost always beneficial with minimal overhead, but also that more powerful partial solvers can be useful.

Based on our experiments, one can make an educated guess which partial solver to select in particular cases; we believe that this selection could even be steered by analysing the parameterised Boolean equation system representing the parity game can be potential future work. It would furthermore be interesting to study (practical)

improvements for the safe attractors, and their use in Zielonka's recursive algorithm. Here, other practical improvements would be to investigate other heuristics to trigger of the partial solvers as hinted on earlier. Another interesting idea was to use the information derived from the partial solving to remove vertices from the *todo* set of the breadth-first search that have already been solved.

There is also future work that is not directly related to this chapter, but is related to extending the symbolic verification techniques present in the mCRL2 toolset. First of all, for the explicit verification the ability to generate counter examples for failing verification as presented in [138] and based on [32] is extremely useful. However, applying this technique to symbolic parity games, although theoretically sound, is not feasible in the way that it is currently implemented. Therefore, making it practical to generate counter examples in this symbolic setting should be considered future work. Finally, since symbolic solving has shown potential in dealing with large state spaces it also makes sense to look into combining it with the decompositional minimisation approach presented in Chapter 3.

Chapter 5

A Thread-safe Term Library

A term is a fundamental concept in mathematics and therefore an equally important common data structure in computing. Many concepts are terms, such as programs, specifications and formulas. Many operations in computing are term transformations, such as compilation. In computer science a term is a far more commonly used concept than structures such as arrays, lists or matrices. This makes it remarkable that terms are not a standard data structure in all common programming languages. However, many functional languages and newer languages such as Haskell [71] and Rust [123] provide so-called algebraic types that can be used to define them.

To our knowledge the first general purpose term library, which is a programming library that facilitates the creation and manipulation of terms, stems from the realm of program transformations. In [5, 84, 17, 15, 16] an ATerm library of so called *annotated terms* has been proposed, which are terms with meta information. Stripping away all additional features from this ATerm format, a very plain and elegant term data structure remains to represent so-called terms.

These terms are defined in the standard way. We start out with a given set of function symbols F where each function symbol $f \in F$ has an arity ar_f. Each constant function symbol, *i.e.*, with arity 0, is a term. Given a function symbol $f \in F$ with $ar_f > 0$, and terms $t_1, \ldots, t_{ar_f}$, the expression $f(t_1, \ldots, t_{ar_f})$ is also a term. These are the only two ways to construct a term.

Note that terms are only a representation of other concepts and some 'constants' could actually represent variables when defining operations on these terms. As an example we can have function symbols $\{0, 1, x, y, +\}$ and terms $0+1$, $x+1$ and $x+y$. The 'constants' x and y would allow for different operations than the constants 0 and 1 as it is natural to define a substitution operation for the constant x, which would be less natural for the constant 0. In a similar way terms with binders can be represented

naturally. For instance in the term $\lambda x.t$ the λ is just a binary function symbol where the first subterm must be a constant (representing a variable), and then operations on this term must take the variable binding into account.

As in the ATerm library, terms are stored in a maximally shared way and once created, terms are stable structures in memory until they are deallocated. This is achieved by a technique introduced already in early implementations of Lisp called *hash consing* [37], where terms of a similar structure are effectively shared using hash table. This leads to a smaller memory footprint, because equal terms are only stored once. Furthermore, comparing terms syntactically is constant time, because two terms are equal *iff* they occupy the same address in memory. Also note that maximal sharing avoids any serialising and deserialising terms to communicate them between threads as done in [5], because this can be achieved by sharing the address of the term. A disadvantage of this approach is that subterms cannot be replaced directly, because these subterms may be shared by other terms. Instead of a hash table we could also consider other associative mappings, such as provided by a CTrie [115], but their reduced memory consumption compared to hash tables often results in reduced performance. The sharing of terms also requires additional bookkeeping to keep track of which terms can still be accessed directly or as part of another term by the program, such that terms can be deallocated from memory when they are no longer in relevant; this process is called *garbage collection*. An alternative is to perform *direct* destruction, where a term is immediately cleaned up when it not relevant. However, often terms with reference count zero could potentially become relevant again within a short time.

With a steadily increasing number of computational cores in computers, it is desirable to have a thread-safe implementation of a term library and this was already stressed in the original publication. Thread-safety means that multiple threads can access and create terms concurrently in the same data structure without it resulting in incorrect behaviour or even program crashes, for example by reading data that has not been properly initialised yet. In our search, we have only found one unpublished thread-safe Java implementation of the ATerm library [90]. Other implementations of thread-safe term (or graph) libraries do exist as part of larger implementations, for example in the Haskell runtime. However, these implementations are not easily extracted and studied by themselves, and also often do not offer the maximal sharing guarantee. Therefore, we study the design and specification of a new thread-safe term library as a programming library by itself.

In our term library terms are static structures in memory, which makes inspecting terms inherently thread-safe. However, the creation and deletion of terms need to take thread-safety into account. This could be achieved by ensuring mutual exclusion of these operations, essentially ensuring that only one thread operates on the term library

at the same time. However, as the operations to create and destroy terms are computationally very cheap, even a small overhead required for thread-safe operations on terms can increase the time of these operations by an order of magnitude. Furthermore, many of algorithms that utilise the term library create and destroy terms frequently and this causes heavy contention on the mutual exclusion variables, which is often undesirable. Therefore, such a naive approach would not work and we need a more fine-grained approach.

Since terms have a tree-like structure it might be expected that concurrent tree algorithms, such as provided by the EXCESS project [135] or the PAM library [129], to provide a possible solution. These tree libraries are focused on manipulating the trees themselves concurrently, for example adding and removing nodes, and rebalancing the tree when required. However, that is different from our terms since these are static structures in memory, and only the creation and destruction of terms can happen concurrently. Instead, we essentially require thread-safety for the following procedures. The creation process consists of performing a lookup in the shared hash table, and actually inserting the element when it has not been found, for the purpose of hash consing. Furthermore, we need a thread-safe mechanism to keep track of which terms are still relevant. We note that destruction only updates the bookkeeping required to keep track of terms and actually only results in the term being removed from memory when it is garbage collected.

First of all, we observe instead of mutual exclusion we only require the behaviour of so-called readers-write locks [31] to ensure that creation and destruction of terms can happen simultaneously (=readers), but garbage collection and hash table resizing (=writer) must be done with exclusive access. We will refer to read access as *shared* access and write access as *exclusive* access. Furthermore, we observe that shared access happens far more often in practice than exclusive access. There are several algorithms for read-dominated readers-write locks in the literature [38, 97, 80, 24]. These algorithms often provide almost no contention to acquire shared access. However, the algorithms do often keep track of additional information to prevent starvation of threads and to deal with high contention scenarios, and these features make them potentially slower due to access to (atomic) variables that are shared between threads. Instead, for our term library we will assume that every thread is performing work related to manipulating terms and makes progress towards completing the task at hand. Furthermore, we assume that the amount of threads closely matches the number of physical cores present in the machine.

Given these two assumptions, we design a new *busy-forbidden protocol* that is also read-optimised, and builds on the ideas of the existing algorithms in taking into account the cache structure of modern processors, but avoids contention on shared access. In this new protocol, obtaining shared access only requires two writes to an atomic variable that is only written to by the current thread and rarely read, and

one read of an atomic variable that is only rarely written to. Therefore, these shared variables are almost always available in the local cache. It is well-known that the design of thread-safe protocols and data structures is quite subtle and certain issues only rarely occur in practice. One example of such a problem is the infamous so-called "ABA problem" [36]. In short, this problem can occur when an algorithm relies on a value A being the same, by for example a compare-and-swap, to indicate that the state has not changed, whereas actually the value A had been temporarily replaced by a value B and thereby invalidating the assumption that no changes took place.

These kind of subtle problems make it essential to formally study the design of these protocols. Therefore, we used the mCRL2 language [63] and corresponding model checking toolset [23] to design both the busy-forbidden protocol and the term library. Using model checking we prove their correctness properties for finite instances before we implemented the prototype. The complete formal specification of the behaviour and the properties is given in Appendix B.

Furthermore, we also require a thread-safe hash table for the purpose of performing hash consing concurrently. Early attempts to create a thread-safe term library led to intriguing wait-free algorithms [76, 47, 48] for the underlying hash table. The assumption was that thread synchronisation was the root cause of performance issues, and this is avoided when algorithms are wait-free. However, the complexity of these algorithms and the amount of accesses to shared variables between threads made these unsuitable in practice. There are in fact thread-safe hash table implementations in the literature [136, 106, 75] that do perform well in practice. However, the implementations of these hash tables proved difficult to be integrated into our prototype and also came with strong assumptions, for example only allowing a single global hash table. Instead, we observe that with some common adaptations, namely essentially introducing a Treiber [134] stack, to an open addressing hash table, inspection and construction can happen concurrently. This worked well in practice and we leave the integration of more sophisticated thread-safe hash table implementations as future work.

In this chapter we present a thread-safe term library and formally verify properties on it. One limitation of the current verification is that we are only able to verify finite instances, but we consider proving the correctness for any number of threads important future work. For the garbage collection we have compared two thread-safe bookkeeping mechanisms for garbage collection, one using atomic operations for reference counting and one that employs explicit thread-local root sets. We have implemented a parallel state space exploration algorithm, which is often the main bottleneck in the verification of large systems, in order to also apply it in practice. Furthermore, we have performed a number of micro-benchmarks as well as a single practical benchmark focused on so-called *strong* scaling, which measures the solution time for increasing amounts of processors given a fixed size problem. This is

reasonable since for state space exploration, our main use case, the sequential part of the problem, for example writing it to disk, also increases with the size of problem. Furthermore, we are generally interested in verifying a given system and property as fast as possible.

The new term library is competitive with our existing sequential term library. Our experiments show that the new term library scales well when terms are heavily shared, as is the case for our practical application. However, when terms are not heavily shared then hash consing and shared access between threads becomes a bottleneck, in which case the maximal sharing guarantee only provides constant time comparison. For the practical case of state space exploration it is already beneficial when only two processors are available, and achieves speed ups of a factor 12 on 16 processors and a factor 16 when using all 32 processors. In general, we find that using thread-local protection sets scales far better than atomic reference counting. Furthermore, we show that the solution with a standard `C++` readers-writer lock, where it is called a shared mutex, and especially the Java implementation of [90] are substantially slower than our implementation with the busy-forbidden protocol. It is intended that the new thread-safe term library will form the heart of the new release of the mCRL2 toolset. The implemented prototype, also as part of the mCRL2 toolset.

Outline In Section 5.1 the specification and implementation of the term data structure, also referred to as term library, is described. In Section 5.2 our new protocol of a readers-writer lock is specified, called the busy-forbidden protocol, and its implementation is discussed. Then in Section 5.3 we introduce the corresponding mCRL2 models and properties, and describe the model checking efforts in detail. Furthermore, in Section 5.4 the term library with the busy-forbidden protocol is compared to several alternative implementations to showcase its performance in practice. Finally, we present a conclusion in Section 5.5.

5.1 Thread-safe Term Library

In [15, 16] a general purpose term library has been proposed called the ATerm library. A term is a very frequently used concept within computer science. The original motivation for terms as a basic data structure came from research in software transformation [84, 17]. The model checking toolset mCRL2 uses terms to represent all internal concepts, such as modal formulas, transition systems and process specifications [63]. We propose a new thread-safe general purpose term library with the behaviour described in the remainder of this section.

5.1.1 External Behaviour

Terms are constructed out of *functions symbols*, or for short *functions*, from some given set F. Each function $f \in F$ has a number of arguments ar_f, generally called the *arity* of f. A function symbol with arity 0 is called a *constant*.

Definition 5.1.1. Let F be a set of function symbols. The set of terms T_F over F is inductively defined as follows:

$$\text{if } f \in F, f \text{ has arity } ar_f \text{ and } t_1, \ldots, t_{ar_f} \in T_F, \text{ then } f(t_1, \ldots, t_{ar_f}) \in T_F.$$

An example of terms are simple numeric expressions. The function symbols are $0, 1, 2, 3, +, *$ where $0, 1, 2, 3$ are constants and $+$ and $*$ have arity 2. An example of a term as a tree structure is given in Figure 5.1.

The term library in [15, 16] allows term annotations, hence the name ATerm, but we do not use this feature. The original ATerm library also supported special terms representing numbers, strings, lists and even 'blobs' containing arbitrary data. We made our own implementation of a term library where besides terms as defined in Definition 5.1.1, there are also facilities for lists and 64-bit machine numbers. As these are in many respects the same as terms constructed out of function symbols, we ignore these additional features for our specification. Instead, the term library provides the following limited set of operations on terms:

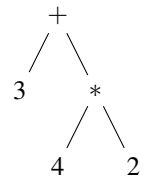


Figure 5.1: The tree representation of the term $3 + 4 * 2$.

Create. Given a function symbol f and terms $t_1, \ldots, t_{ar_f}$ construct a term $f(t_1, \ldots, t_{ar_f})$. This operation can fail when there is not enough memory.
Destroy. Indicate that a term t will not be accessed anymore by this thread. Terms that are not accessed by any thread must ultimately be garbage collected.
Argument. Obtain the i-th subterm t_i of a term $f(t_1, \ldots, t_{ar_f})$.
Function. Obtain the function symbol f of a term $f(t_1, \ldots, t_{ar_f})$.
Equality. For terms t and u determine whether t and u are equal. Note that due to maximal sharing this operation only requires constant time.

The typical usage pattern of terms is that they are visited very often by inspecting arguments or function symbols. Creation of a term is also a very frequent operation, but in the majority of cases (more than 90 percent) a term is created that already exists in memory and only a lookup has to be performed. The garbage collection and hash table resizes are performed internally and only triggered infrequently compared to all the other operations.

5.1.2 Behavioural Properties

Our term library guarantees the following properties, checked using model checking on models of finite instances of our term library, see Section 5.3.3.

1. A term and all its subterms remain in existence at exactly the same address, with unchanged function symbol and arguments, as long as it is not destroyed.
2. Two stored terms t_1 and t_2 always have the same non-null address iff they are equal.
3. Any thread that is not busy creating or destroying a term, can always initiate the creation of a new term or the destruction of any term that this thread has access to.
4. Any thread that started creating a term or destroying a term, will eventually successfully finish this task provided there is enough memory to store one more term than those that are accessible. But it is required that other threads behave fairly, in the sense that they will not continually create and destroy terms or stall other threads by busy waiting.

Note that the properties above imply some notion of deallocation in the sense that if a thread makes and destroys terms, and these are not deallocated, at some point no new terms can be created due to a lack of memory and in that case property 4 above would be violated.

5.1.3 Implementation

Terms are immutable maximally shared tree structures in memory. This means that if two (sub)terms are the same, they are represented by the same object in memory, as shown in Figure 5.2. Terms in our term library can be copied, constructed and accessed in parallel. Note that *copying* of terms is observably the same as creating a term with the same function symbol and arguments, but can be implemented more efficiently.

Figure 5.2: The tree representation of $f(c, f(c, c))$.

By storing terms as maximally shared trees, the only nontrivial operations on terms are the creation of a new term and the destruction of an existing term. Due to the immutable nature of terms it is not possible to simply replace a subterm of a term. If a subterm must be changed, the whole surrounding term must be copied. From the perspective of the programmer this means that we indeed have no operations to modify terms. However, this immutability makes these terms

very suitable for parallel programming. Threads can safely traverse protected terms in memory as they can be sure that they will not change.

The maximal sharing is implemented in the term library using *hash consing*, which is implemented using a hash table as follows. Whenever a term with function symbol f and arguments $t_1, \ldots, t_{ar_f}$ is created, a lookup is performed on the hash table to determine whether $f(t_1, \ldots, t_{ar_f})$ already exists. If this is the case, its current address is returned. Otherwise, a new term $f(t_1, \ldots, t_{ar_f})$ is allocated and inserted in the hash table, and its address is returned.

Terms that are accessible, either directly or as subterm, by thread must be identified as such in some way, which we call *protecting* the term as it prevents the term from being garbage collected. There are essentially two fundamentally different ways to keep track of the protected terms. The first one is to keep a *reference count* in each created term, counting how many references there are to the term. When the term is created, the reference count is incremented, and when a term is destroyed, the reference count is decremented. If the term has reference count 0, its address is freed up, also called deallocated, during garbage collection.

The other way to protect terms is to maintain a set of addresses where terms are being stored, which is often referred to as *root* set and we refer to these addresses as variables. A *mark-and-sweep* garbage collection algorithm can be used where the subterms of protected terms are recursively marked, and all unmarked terms are finally freed up. In our current implementation garbage collection is performed by a single thread. There are also parallel garbage collection algorithms where creation and destruction can happen simultaneously, for example in [48]. These algorithms are generally complex and in our use case we are not bound by real-time constraints. In our current benchmarks we also find that garbage collection requires only a small fraction of the total time. However, performing the marking and sweeping phases using multiple threads simultaneously could be potentially beneficial to further improve scalability, so we consider this to be potential future work.

In the parallel setting we must ensure so-called sequential consistency when changing reference counts since these changes must always be immediately visible in other threads. Sequential consistency here means that these updates must behave atomically as if interleaved one after each other. Changing reference counts is a very frequent operation and often leads to cache contention as the reference counts are accessible by all threads simultaneously. Operations on the protection sets are more complex than changing a reference count, but they can be performed locally in a thread and only require shared access to the readers-writer lock. Furthermore, variables in the root set can be reused when assignments are performed on them, whereas references count must be updated on every assignment, so depending on the structure of the program there are far less changes to the root set when compared to reference counts.

We use a hash table with *open hashing*, which is also referred to as separate

chaining. Here, we use a linked list as a so-called bucket list to deal with collisions of terms in the table. If the term does not exist, it is added using a compare and swap operation to the bucket list of the appropriate entry of the hash table. If in the mean time another thread creates the same term, the compare and swap fails, informing the thread that it has to inspect the hash table again to find out whether the term came into existence. This is the Treiber's stack [134]. We remark that it is safe to use because terms are only deleted from the hash table during garbage collection, and during garbage collection no new terms are allowed to be constructed. This is a fairly simplistic parallel hash table implementation, but our prototype shows that this works well enough in practice. Further research into more complicated, but potentially more effective, hash table implementations is left as future work.

Accessing terms during garbage collection and resizing, also called rehashing, the hash table is perfectly safe. However, it is not allowed to create or copy terms while garbage collection or resizing is being performed. Therefore, we require a mutual exclusion protocol where either multiple threads can create and copy terms simultaneously, which we call the *shared* tasks, or one thread can be involved in garbage collection or rehashing, which is called the *exclusive* task. This is the behaviour of a readers-writer lock [117] where multiple readers or at most one writer can access a shared resource, where reading is the shared task, and writing is the exclusive task. As we remarked earlier creating and copying terms is done very frequently, and thus shared access must be very cheap and exclusive access can be expensive. For this purpose, we developed a completely new protocol, called the *busy-forbidden* protocol, to serve our needs.

Next, we present the pseudocode for the procedures described so far. Table 5.1 contains the code for creating and destroying terms. The functions `protect`, `unprotect` and `protected` refer to the protection mechanisms described previously, in which `protected`(t) will return true if and only if the term t is protected by some thread. In this code `enter_shared`, `leave_shared`, `enter_exclusive` and `leave_exclusive` are part of the busy-forbidden protocol described in the next section.

The function `h` is a hash function that takes a function symbol f, and subterms $t_1, \ldots, t_n$, and calculates a position in the table. The hash table is represented by the variable *table*. It has an array of buckets *buckets* where every element b contains an atomic pointer *b.top* to a so-called node list that allows atomic loads and an atomic compare-and-swap operation `cmpswap`, which returns true iff it is successful. The node list is a singly-linked list of *Node* objects that have a *head* and *tail* pointing to the head and tail of the list respectively.

```
1  create(thread p, symbol f, subterms t_1,...,t_n)
2  {
3    enter_shared(p)
4    hash ← h(f,t_1,...,t_n)
5    bucket ← table.buckets[hash % |table.buckets|]
6    t ← insert(bucket,f,t_1,...,t_n)
7    protect(p,t)
8    leave_shared(p)
9    return t
10 }
11
12 insert(bucket b, symbol f, subterms t_1,...,t_n)
13 {
14   old_head ← b.top
15   node ← b.top
16   do
17   {
18     if node.head = f(t_1,...,t_n)
19       return node.head
20     node ← node.tail
21   }
22   while (node ≠ NULL)
23   t ← construct f(t_1,...,t_n)
24   if not cmpswap(b.top,old_head,Node(t,old_head))
25   {
26     destruct t
27     return insert(b,f,t_1,...,t_n)
28   }
29   return t
30 }
```

```
destroy(thread p, term t)
{
  unprotect(p);
  if necessary()
    GC(p)
}

GC(thread p)
{
  enter_exclusive(p);
  forall t ∈ table
  {
    if not protected(t)
      remove t;
  }
  leave_exclusive(p);
}
```

Table 5.1: Pseudocode for the thread-safe term library.

Garbage collection has a *necessary* condition to define when it should be called. In principle it can be called after every destroy, but in practice heuristics can be used to only have constant overhead per created term. This can be achieved by initialising a counter with the current number of terms at the end garbage collection that is decremented on every term creation, and only garbage collect when the counter reaches zero again.

Using an mCRL2 model of the behaviour of the term library, the behavioural

properties mentioned in Section 5.1.2 have been model checked for finite instances. This is described in more detail in Section 5.3.3.

5.2 Busy-Forbidden Protocol

In this section we study the busy-forbidden protocol in more detail. As mentioned before, the busy-forbidden protocol is designed for the situation where shared access is frequent whereas exclusive access is infrequent.

5.2.1 External Behaviour

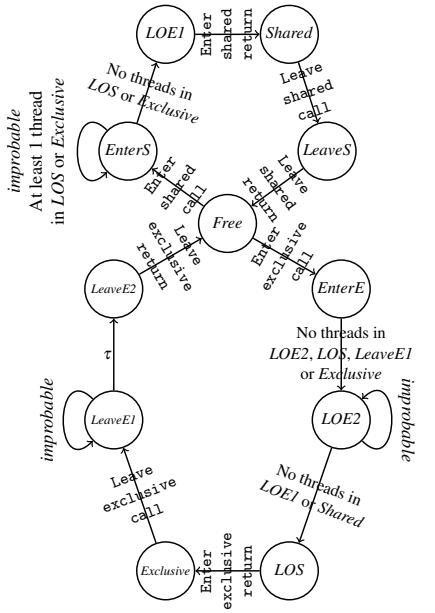
We first look at the external behaviour of this protocol. As indicated above, threads can request for shared or exclusive access by calling one of the two functions `enter_shared` and `enter_exclusive`. The functions starting with `leave` are used to indicate that access is no longer required.

We make the external behaviour more precise by modelling it as a state automaton, whose behaviour is equivalent to the mCRL2 specification used for verification later on. From the perspective of a single thread, the behaviour is depicted in Figure 5.3. The function calls are modelled by actions named `Enter/Leave shared/exclusive call` respectively. Returning from the function is modelled by actions with names ending with `return`. This protocol guarantees that at most one thread can be in state *Exclusive* and if a thread is in state *Exclusive*, no thread is in state *Shared*, and vice versa, if there are threads in state *Shared*, then there is no thread in the state *Exclusive*.

The center state, labelled *Free*, indicates that the thread is not involved in the protocol, *i.e.*, it is outside the shared and exclusive sections. Following the arrows in a clockwise fashion, a thread obtains access. In the state *EnterS* the thread requested shared access, and it will get it when there are no threads in the states *LOS* or *Exclusive*.

Our main goal was to ensure that the behaviour of Figure 5.3 for multiple threads is so-called divergence-preserving branching bisimilar to the implementation that is described later on [57, 58]. The reason is that this equivalence relation preserves not only safety but also most liveness properties. This allows us to replace the large implementation model, which contains a lot of intrinsic details, by the much smaller specification model when verifying the term library itself.

Divergence preserving branching bisimulation does not remove τ-loops, *i.e.*, loops of internal actions, compared to typical branching bisimulation. There are various self-loops introduced in the model to account for infinite loops that are unlikely, or even impossible under fair scheduling, and to ensure equivalence of behaviour to the implementation model. Therefore, these actions are marked *improbable*, and are seen as internal actions.



LOE1	There are no threads in or able to enter *Exclusive*.
Shared	Shared access. No concurrent access to *Exclusive* possible.
EnterS	Entering shared.
LeaveS	Leaving shared.
Free	The thread is outside any exclusive or shared section.
LeaveE2	Leaving exclusive.
EnterE	Entering exclusive.
LeaveE1	Leaving exclusive. No threads in or able to enter *Exclusive*.
LOE2	There are no threads in or able to enter *Exclusive*.
Exclusive	Exclusive access. There are no threads in or able to enter *Exclusive* or *Shared*.
LOS	No threads in or able to enter *Exclusive* or *Shared*.

Figure 5.3: The external behaviour of the busy-forbidden protocol.

5.2.2 Implementation

The implementation code for entering and leaving the exclusive sections is presented in Table 5.2. The busy-forbidden protocol is implemented by assigning to each thread two atomic flags, called *busy* and *forbidden*. The *busy* flag indicates that the current thread is in its shared section and can only be written to by this thread. The flag *forbidden* indicates that some thread is having exclusive access. There is one busy and one forbidden flag per thread.

```
1  enter_shared(thread p)
2  {
3      p.busy ← true;
4      while p.forbidden
5      {
6          p.busy ← false;
7          if mutex.timed_lock()
8          {
9              mutex.unlock();
10         }
11         p.busy ← true;
12     }
13 }
14
```

```
1  enter_exclusive(thread p)
2  {
3      mutex.lock();
4      while exists thread q with
5              ¬q.forbidden
6      {
7          select thread r
8          r.forbidden ← true;
9          if r.busy or sometimes
10         {
11             r.forbidden ← false;
12         }
13     }
14 }
```

```
1  leave_shared(thread p)
2  {
3      p.busy ← false;
4  }
5
6
7
8
9
10
11
12
13
```

```
1  leave_exclusive(thread p)
2  {
3      while exists thread q with
4              q.forbidden
5      {
6          select thread r
7          usually do
8              r.forbidden ← false;
9          sometimes do
10             r.forbidden ← true
11     }
12     mutex.unlock();
13 }
```

Table 5.2: Pseudocode description of the busy-forbidden protocol.

Besides the flags there is one generic mutual exclusion variable, called *mutex*. The variable *mutex* can not only be locked and unlocked, but also provides a timed lock operation *timed_lock*(). It tries to lock the mutex, and if that fails after a certain time, it returns false without locking it. This timed mutex is only important for performance, and can also be omitted altogether.

When entering the shared section, a thread generally only accesses its own *busy* and *forbidden* flags as *forbidden* is almost always false. These flags are only rarely accessed by other threads and therefore almost always available in the local cache of the core executing the thread. In the rare case when the *forbidden* flag is set, this

thread backs off using *mutex* to try again later. In principle the while-loop can be iterated indefinitely, giving rise to the internal loop in state *EnterS* in the specification. Leaving the shared section only consists of setting the *busy* flag of the thread to false.

Accessing the exclusive section is far more expensive. By using *mutex*, mutual access to the exclusive section is obtained. Subsequently, the *forbidden* flag for each thread p is set to true, unless the *busy* flag of thread p is set, as in this case the *forbidden* flag must be set to false again. There is a non immediately obvious scenario where one thread refuses to leave the shared section, and two other threads p_2 and p_3 want to access the shared, respectively, exclusive section. Thread p_3 cannot obtain exclusive access, but hence should not indefinitely block shared access for p_2. Hence, p_3 must set the *forbidden* flag of p_2 to false if *busy* of p_1 is true. Without the **sometimes** part the implementation is not divergence preserving bisimilar to the specification, because then *r.busy* being false on line 9 leads to a state without an internal loop, which does not occur in the specification, if all forbidden flags are set. Without the **sometimes** part, a matching specification would become substantially more complex exhibiting exactly when which *forbidden* flag is set, rendering the specification far less abstract.

When leaving the exclusive section a thread resets all *forbidden* flags of the other threads. Again, if this is done in a predetermined sequence the divergence preserving branching bisimilar external behaviour becomes very complex, as this sequence has an influence on the precise sequence other threads can enter the shared section. By resetting and sometimes even setting the *forbidden* flag, a comprehensible provably equal external behaviour is obtained, although it leads to another loop of internal actions in the specification. Practically, re-resetting is not needed, and certainly not for our term library. However, it is interesting to further investigate the optimal use of the timing of *mutex* in `enter_shared`, as well as the optimal rate of occurrence of the **sometimes** instructions for generic uses of the busy-forbidden protocol.

5.2.3 Behavioural Properties

We also formulate a number of natural requirements that should hold for this protocol. These requirements have been formulated as modal properties and verified using model checking.

1. There should never be more than one thread present in the exclusive section.
2. There should never be a thread present in the exclusive section while one or more threads are present in the shared section.
3. When a thread requests to enter the shared section, it will be granted access within a bounded number of steps, unless there is another thread in the exclusive section.

4. When a thread requests to enter the exclusive section, it will be granted access within a bounded number of steps, unless there is another thread in the shared or in the exclusive section.

5. When a thread requests to leave the exclusive/shared section, it will leave it within a bounded number of steps.

6. A thread not in the exclusive or shared section can instantly start to enter the exclusive or shared section.

For properties 3, 4, and 5 granting access and leaving can be indefinitely postponed if other threads are entering and leaving exclusive and shared sections, or when other threads are in the while loops, continuously writing forbidden and busy flags. This means that these properties only hold under fair scheduling assumptions for the threads.

5.3 Modelling and Verifying the Algorithms

We have made models of the busy-forbidden protocol and the thread-safe term library in the process modelling language mCRL2 [63] based on the pseudocode in both Table 5.1 and 5.2. Furthermore, we have formalised the informal requirements listed in Sections 5.1.1 and 5.2.3 in the modal mu-calculus, and verified them using the mCRL2 toolset [23].

The models and formulas can be found in Appendices B.1 and B.2, respectively. Due to the nature of model checking, we only verify the models for finite instances. We repeatedly found that when protocols or distributed systems are erroneous, the problems already reveal themselves in small instance [64]. After verification succeeded on finite instances we have not encountered any issues during the execution of the protocol in practice. The protocol and library have not been proven correct in general for any number of threads and terms. Proving the specification and implementation of the busy-forbidden protocol equal is conceivable using a variant of the Cones and Foci method [65, 46], and this is considered future work. Unfortunately, we do not know of any effective method to prove modal formulas on models with a complexity such as ours automatically for any number of threads and terms, and consider this an important direction of research.

5.3.1 The mCRL2 Language and Modal Formulas

The mCRl2 language is a modelling language based on CCS (Calculus of Communicating Processes) [103] and ACP (Algebra of Communicating Processes) [2]. Note that the process algebra with multi-actions that we have defined in Chapter 3 was

derived from the mCRL2 language, so the following concepts will be very similar. The behaviour of processes is given by the atomic actions that are observed, which in our case represent the function calls and access to (global) variables, and the special action τ represents *internal* behaviour. We again assume the existence of standard data types as *Bool* representing $\mathbb{B}$ and *Nat* representing $\mathbb{N}$ with standard operators.

Processes can be sequentially composed using the dot ('.') operator. Alternative composition, where non-deterministically one of the options can be chosen, is denoted using a plus ('+'), or in general by a summation ('Σ'), and parallel composition is denoted by $\|$. Using the **comm** and **allow** operators, the synchronisation of parallel components can be specified to achieve message passing or communication. For example, we can specify that a $store_p$ action can only occur iff a corresponding $store_f$ action can occur with the exact same values for its arguments. Furthermore, an if-then-else is written as $c \rightarrow p \diamond q$ where the behaviour of process p is executed iff c is true, and otherwise the behaviour of process q takes place.

Recursive behaviour is denoted using equations of the form $X = p$, *e.g.*, $X = a\,.\,X$ is the process that can perform an infinite number of a actions. The process variables X can have data parameters. For example, a counter process can be defined as $C(n : Nat) = up\,.\,C(n+1)$. We also allow functions as process parameters, for example the process variable $Y(m : Nat \rightarrow Bool)$ uses a mapping m from natural numbers to booleans. The function update $m[n \mapsto b]$ specifies that $m[n \mapsto b](k)$ equals b if $k = n$ and equals $m(k)$ otherwise.

We also have the modal mu-calculus to specify behaviour requirements of these processes. Formulas consist of conjunctions ($\wedge$), disjunctions ($\vee$), implications ($\rightarrow$), negations ($\neg$), predicates and quantifiers. The diamond modality $\langle a \rangle \phi$ is valid iff an action a can occur after which ϕ holds. The dual box modality $[a]\phi$ holds iff after every possible a action ϕ holds. The actions inside these modalities are generalised to regular formulas. First, an action *formula* represents a set of multi-actions and is either a multi-action, a predicate over multi-actions with quantifiers over data, or the usual set operations union ($\cup$), intersection ($\cap$) and complement ($\overline{U}$). Next, *regular* formulas are the typical sequential composition ($.$), choice ($+$) and Kleene star (*). For example: the formula $\langle a\,.\,(b \cup c) \rangle \mathsf{true}$ only holds if we can either do an a action followed by a b or c action. Furthermore, we also have sets of actions where true represents the set of all actions with the typical set union ($\cup$), intersection ($\cap$) and complement ($\overline{U}$). An often occurring pattern is $[\mathsf{true}^*]\phi$ expressing that ϕ must hold in all states reachable via a sequence of actions.

We can also write recursive formulas using the minimal fixed point operator $\mu X.\phi$ and the maximal fixed point operator $\nu X.\phi$. For example the maximal fixed point operator can be used to construct the formula $\nu X.\langle a \rangle X$, which expresses that we must be able to perform action a after which the same formula still holds. Thus this formula only holds if we can perform an infinite amount of a actions.

A noteworthy fixed point construction, used in several properties, is the following:

$$\nu X.\mu Y.([\overline{a \cup b}]Y \wedge [b]X \wedge \langle \mathsf{true}^* \,.\, a \rangle \mathsf{true}) \tag{5.1}$$

Here we state that an a action must always be able to occur within a finite amount of steps, unless a b action occurs infinitely often. This construction is useful for properties in which we state that something may eventually happen given fair scheduling (the b action does not occur infinitely often).

The fixed point operators also allow us to pass on parameters in the same way we can do for process variables. This allows us, for example, to keep track of the number of times that a given action has occurred, *e.g.*, given a system with the actions *in* and *out*, we can state that each *out* action needs a corresponding *in* action using the following fixed point:

$$\nu X(n{:}\mathbb{N} = 0).[\overline{in \cup out}]X(n) \wedge [in]X(n+1) \wedge [out](n > 0 \wedge X(n-1)).$$

Here n keeps track of the difference between the amount of *in* and *out* actions. We use $n{:}\mathbb{N} = 0$ to state that n is a natural number and is initially 0. The left-hand side of the conjunction $(n > 0 \wedge X(n-1))$ states that an *out* action may only occur if n is greater than 0.

5.3.2 Modelling and Verifying the Busy-Forbidden Protocol

The process specification given in Table 5.3 exactly matches the external behaviour (or specification) shown in Figure 5.3. We define P to be the (finite) set of threads and we define S to be a data set representing the set of states:

$$S = \{\ \mathit{Free}, \mathit{EnterS}, \mathit{LOE1}, \mathit{Shared}, \mathit{LeaveS}, \mathit{EnterE}, \mathit{LOE2}, \mathit{LOS}, \mathit{Exclusive}, \mathit{LeaveE1}, \mathit{LeaveE2}\ \}$$

Initially, $s(p) = \mathit{Free}$ for all $p \in P$. The conditions for performing transitions are the same as the conditions in the diagram of the external behaviour.

The implementation model is described in great detail in Appendix B.1, but generally follows the pseudocode presented in Table 5.2 translated into mCRL2. Both models use the eight externally observable actions mentioned earlier, such as `enter_shared_call` and `enter_shared_return`. Note that the *mutex.timed_lock*() statement is omitted as it is only important for performance and not for correctness. We have shown for finite instances up to seven threads that these are divergence preserving branching bisimilar when abstracting the 'improbable' actions and performing non-deterministic choice for the 'sometimes' statement.

As the next step, we transformed the six requirements discussed in Section 5.2.3 into modal logic formulas, and verified them on the specification. Note that these

$$
\begin{array}{l}
BF(s : P \to S) = \\
\sum_{p:P} . (\\
\quad (s(p) \approx \mathit{Free}) \\
\quad \to \texttt{enter\_shared\_call}(p) \,.\, BF(s[p \mapsto \mathit{EnterS}]) \\
+ \ (s(p) \approx \mathit{EnterS}) \\
\quad \to ((\neg \exists_{p':P} .\ s(p') \in \{\mathit{LOS}, \mathit{Exclusive}\}) \to \tau \,.\, BF(s[p \mapsto \mathit{LOE1}]) \\
\quad \diamond \ \mathit{improbable} \,.\, BF(s)) \\
+ \ (s(p) \approx \mathit{LOE1}) \\
\quad \to \texttt{enter\_shared\_return}(p) \,.\, BF(s[p \mapsto \mathit{Shared}]) \\
+ \ (s(p) \approx \mathit{Shared}) \\
\quad \to \texttt{leave\_shared\_call}(p) \,.\, BF(s[p \mapsto \mathit{LeaveS}]) \\
+ \ (s(p) \approx \mathit{LeaveS}) \\
\quad \to \texttt{leave\_shared\_return}(p) \,.\, BF(s[p \mapsto \mathit{Free}]) \\
+ \ (s(p) \approx \mathit{Free}) \\
\quad \to \texttt{enter\_exclusive\_call}(p) \,.\, BF(s[p \mapsto \mathit{EnterE}]) \\
+ \ (s(p) \approx \mathit{EnterE} \wedge \neg \exists_{p':P} .\ s(p') \in \{\mathit{LOE2}, \mathit{LOS}, \mathit{Exclusive}\}) \\
\quad \to \tau \,.\, BF(s[p \mapsto \mathit{LOE2}]) \\
+ \ (s(p) \approx \mathit{LOE2}) \\
\quad \to \mathit{improbable} \,.\, BF(s) \\
+ \ (s(p) \approx \mathit{LOE2} \wedge \neg \exists_{p':P} .\ s(p') \in \{\mathit{LOE1}, \mathit{Shared}\}) \\
\quad \to \tau \,.\, BF(s[p \mapsto \mathit{LOS}]) \\
+ \ (s(p) \approx \mathit{LOS}) \\
\quad \to \texttt{enter\_exclusive\_return}(p) \,.\, BF(s[p \mapsto \mathit{Exclusive}]) \\
+ \ (s(p) \approx \mathit{Exclusive}) \\
\quad \to \texttt{leave\_exclusive\_call}(p) \,.\, BF(s[p \mapsto \mathit{LeaveE1}]) \\
+ \ (s(p) \approx \mathit{LeaveE1}) \\
\quad \to \mathit{improbable} \,.\, BF(s) \\
+ \ (s(p) \approx \mathit{LeaveE1}) \\
\quad \to \tau \,.\, BF(s[p \mapsto \mathit{LeaveE2}]) \\
+ \ (s(p) \approx \mathit{LeaveE2}) \\
\quad \to \texttt{leave\_exclusive\_return}(p) \,.\, BF(s[p \mapsto \mathit{Free}]) \\
\quad) \\
\end{array}
$$

Table 5.3: Specification of the busy-forbidden protocol corresponding to Figure 5.3.

$$
\begin{array}{ll}
1 & \nu X(n_{shared} : Nat = 0,\ n_{exclusive} : Nat = 0). \\
2 & \quad (\forall_{p:P}.[\texttt{enter\_shared\_return}(p)]X(n_{shared}+1, n_{exclusive})\ ) \\
3 & \wedge\ (\forall_{p:P}.[\texttt{enter\_exclusive\_return}(p)]X(n_{shared}, n_{exclusive}+1)\ ) \\
4 & \wedge\ (\forall_{p:P}.[\texttt{leave\_shared\_call}(p)]X(n_{shared}-1, n_{exclusive})\ ) \\
5 & \wedge\ (\forall_{p:P}.[\texttt{leave\_exclusive\_call}(p)]X(n_{shared}, n_{exclusive}-1)\ ) \\
6 & \wedge\ [\\
7 & \quad \overline{(\exists_{p:P}.\texttt{enter\_shared\_return}(p)\ )} \\
8 & \quad \cap\ \overline{(\exists_{p:P}.\texttt{enter\_exclusive\_return}(p)\ )} \\
9 & \quad \cap\ \overline{(\exists_{p:P}.\texttt{leave\_shared\_call}(p)\ )} \\
10 & \quad \cap\ \overline{(\exists_{p:P}.\texttt{leave\_exclusive\_call}(p)\ )} \\
11 & \quad]X(n_{shared}, n_{exclusive}) \\
12 & \wedge\ \neg(n_{exclusive} > 0 \wedge n_{shared} > 0) \\
13 &
\end{array}
$$

Table 5.4: The modal formula for property 2: "There should never be a thread present in the exclusive section while one or more threads are present in the shared section".

properties are preserved by divergence preserving branching bisimulation, so verifying these properties again for the implementation model is not necessary. We only discuss property 2 as an illustration of what such formulas look like, and the other formulas are presented in Appendix B.1.1. The informal description of the property reads:

2. There should never be a thread present in the exclusive section while one or more threads are present in the shared section.

The corresponding modal formula is shown in Table 5.4. We use a maximal fixpoint with two data parameters, namely n_{shared} and $n_{exclusive}$, both initially being 0. The argument n_{shared} indicates the number of threads present in the shared section, and $n_{exclusive}$ the number in the exclusive section. On line 2 through 5, we keep track of the amount of threads present in each section, updating the variables after each respective action. On line 6 through 11, we say that our variables stay the same, after any action that is not one of the four aforementioned actions. Finally, on line 12, we say that threads are only allowed to be either present in exclusive or are present in shared.

These properties were verified for up to 7 threads on the specification model. For comparison, the specification model has about three million states and the implementation model about 11 billion states. We uncovered a number of issues in earlier versions and obtained various insights during development while doing the verification for 2 and 3 threads. The verification with more threads, although increasingly time consuming, did not lead to any additional insight.

5.3.3 Modelling and Verification of the Term Library

We also modelled the implementation of the thread-safe term library following the pseudocode shown in Table 5.1.

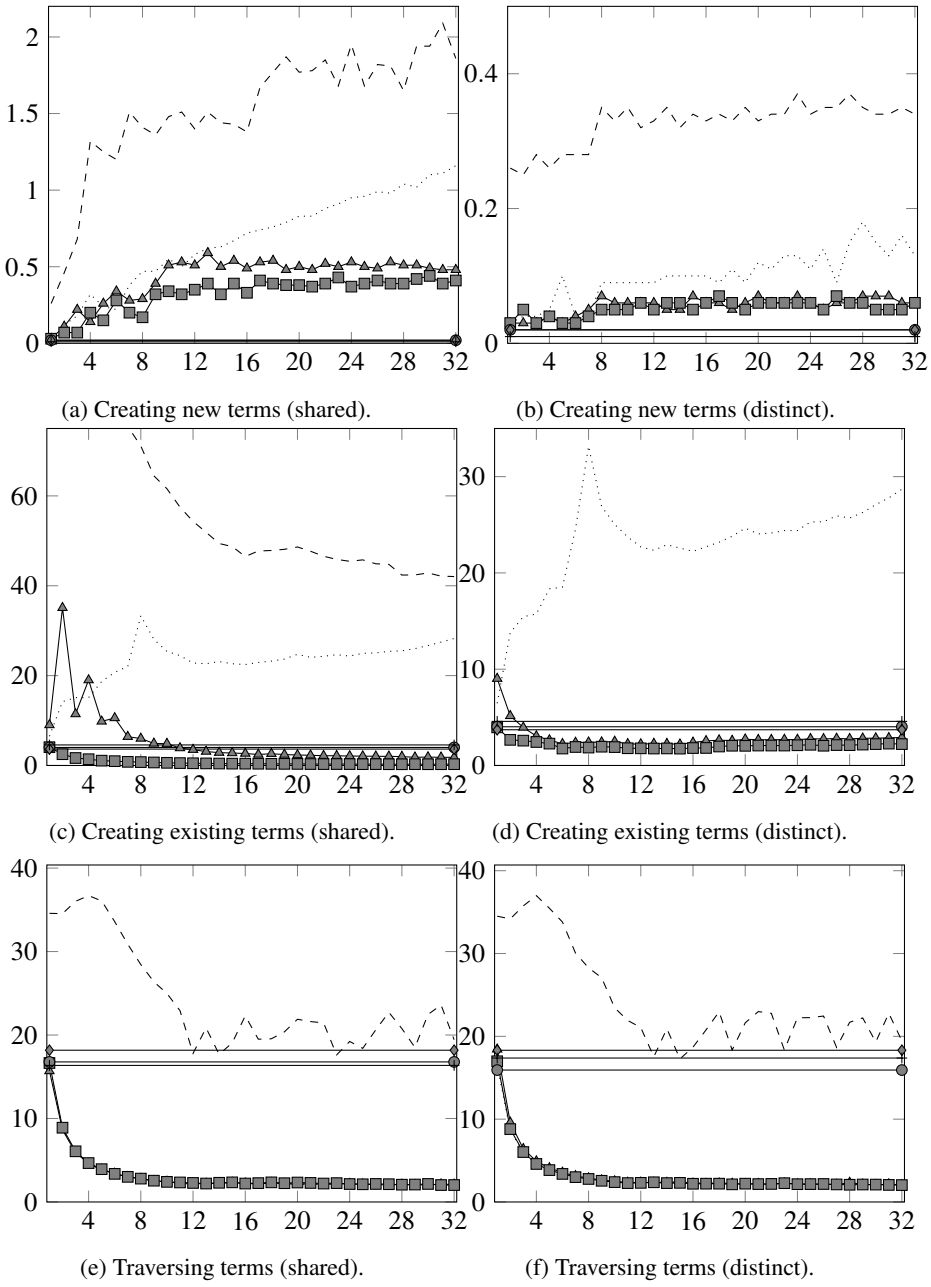
The model uses four externally observable actions, such as `create_call` and `create_return`, to represent calling and returning from either the `create` or the `destroy` functions, specified in Section 5.1.1. The action `create_return`(t, a, p) represents a `create`(t) call by thread p returning the address a where t is stored.

To reduce complexity of verification we assume a correct hash table implementation, which means that we do not model the behaviour of the bucket list and hash functions explicitly. Instead the hash table is modelled as a simple associative array, with atomic *contains* and *insert* operations. The model is primarily concerned with the thread-safe creation and garbage collection of terms, and therefore the typical term structure, where terms contain subterms, is also not part of the model, and thus we model only constant terms.

The four properties discussed in Section 5.1.2 were also translated into modal logic and verified using model checking. We have verified that these properties hold for up to 3 threads, using 3 different terms and 4 possible addresses. We use the specification of the busy-forbidden protocol in this model and this resulted in a state space of 800 million states The use of the implementation model for the busy-forbidden protocol would lead to a far larger state space. Furthermore, we were unable to verify our properties on larger state spaces as they became too large. For example the state space of the aforementioned setup with 4 threads instead of 3 already has 129 billion states, which could only be inspected using symbolic representations using the techniques described in Chapter 4.

5.4 Performance Evaluation

We have implemented a sequential and a parallel version of the term library in `C++` as part of our mCRL2 toolset [23]. These implementations are almost identical except for the synchronisation primitives added to the parallel version where necessary. For the parallel version we compare the busy-forbidden protocol to the readers-writer lock implementation present in the `C++` standard library, where it is called a shared mutex. Furthermore, we have implemented both reference counting and root protection sets as garbage collection strategies in both implementations for comparison. We compare these implementations with the sequential term library as it is currently used in the mCRL2 toolset [63] and to both a thread-safe [90] and a sequential Java implementations of the ATerm library. All reported measurements are the average of five runs with an AMD EPYC 7452 32-Core processor, unless stated otherwise.



(a) Creating new terms (shared).

(b) Creating new terms (distinct).

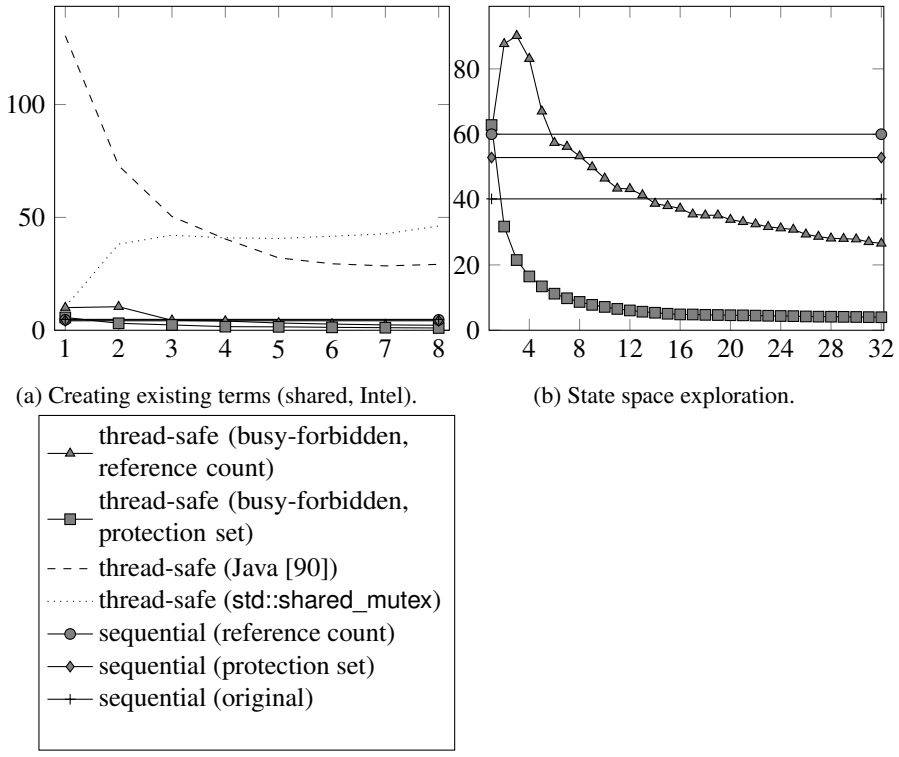
(c) Creating existing terms (shared).

(d) Creating existing terms (distinct).

(e) Traversing terms (shared).

(f) Traversing terms (distinct).

Figure 5.4: The experimental results.



(a) Creating existing terms (shared, Intel).

(b) State space exploration.

(c) Legend for all plots in Figures 5.4 and 5.5.

Figure 5.5: Additional benchmarks and legend.

The results are listed in the plots in Figures 5.4 and 5.5, and for completeness we also present the exact values as tables in Appendix B.3. In these plots the y-axis indicates the wall clock time in seconds and the x-axis the number of threads (indicates as #*threads*). The legend for all plots is presented in Table 5.5c, with the following explanation. Triangles are the parallel reference counted implementation and the squares the parallel protection set implementation. For the sequential versions we have circles for the reference counted variant, diamonds for the protection set variant and plusses for the original implementation. Finally, the dashed line indicates the thread-safe Java implementation and the dotted line is our thread-safe implementation where the busy-forbidden protocol has been replaced by the standard shared lock. This

last implementation uses protection sets. Furthermore, we note that for the sequential implementations the lines are extended horizontally for easier comparison.

In Figure 5.4 we report on three experiments, one per row, designed to obtain insight in how the new library performs for certain micro-benchmarks. In the left column all threads access the same shared term, whereas in the right column each thread operates on its own term, but due to hash consing these terms are all stored in the globally shared hash table.

In Figure 5.4 (a) we measure how expensive it is to create a term in parallel that does not yet exist, which is uncommon in practice. Each thread creates a term $t_{400\,000}$ defined as follows. The term t_0 is equal to a constant c and t_i is $f(t_{i-1}, t_{i-1})$ for a function symbol f of arity two, which is the most common arity used in practice. Note that due to sharing, this term consists of 400 001 term nodes. In (b) each thread creates a term $t_{400\,000/\#threads}$ instead where c is a unique constant for each thread, creating a total of $400\,000 + \#threads$ term nodes. In Figures 5.4 (c) and (d) we measure the time it takes to create $1000/\#threads$ instances of the terms used in (a) and (b) respectively. This measures the time it requires to create terms that are already present in the hash table, and this essentially boils down to a hash table lookup.

In the lower diagrams, *i.e.*, (e) and (f), we measure the time required to perform $1000/\#threads$ breadth-first traversals on a term t_{20}, where again in (f) these terms are unique per thread. The traversals do not employ the shared structure, hence $2^{21} - 1$ terms are visited per traversal.

First, we observe that our term library completely outperforms the Java implementation, which is generally in line with the expectation that efficient memory management and offline compilation in `C++` outperforms Java implementations. In diagram (d) the Java results are even left out as Java consistently requires more than 100 seconds. Furthermore, we observe that the shared lock is slower than the busy-forbidden protocol in the equal comparison where both use protection sets in all cases. Similarly, we can observe that the reference counted variant is overall slower than variant using a protection set. For traversing terms no locking is required, and therefore, no difference is observed with the dotted line hidden under the line of boxes.

The thread-safe term library outperforms the parallel Java version and the implementation with a standard shared lock for all cases for the protection set variant. The thread-safe term library scales poorly for the case where new terms are created that do not yet exist in the hash table. Similarly, there is only a speed up of about two on 16 threads, and a slightly worse speed up for 32 threads, for the case where distinct existing terms are created. However, we measure that about 90 percent of terms already exist in the hash table in our practical applications. For this use case the thread-safe term library with the busy-forbidden protocol and protection sets scales well with the number of threads, achieving a speed up of 10 on 32 threads for creating existing shared terms, and is competitive with the sequential implementation. For

term traversal it scales well up to 8 threads, after which memory accesses seemingly become the bottleneck.

We observe in Figure 5.4 (c) that the reference counting implementation for a few threads is unexpectedly inefficient. In order to understand this, we retried the experiments on an Intel i7-7700HQ processor, reported in Figure 5.5 (a). Here, none of the anomalies occur, and notably, Java even outperforms the standard shared lock implementation with more threads. Unfortunately, we have no explanation for this, but it is in line with our general observation that compiler and processor can have a large influence on the benchmark results.

The dedicated benchmarks are promising, but in order to get insight in the behaviour of the term library in practical situations, we incorporated the term library in the mCRL2 toolset and used it to generate the state space of the 1394 firewire protocol [98]. These results are shown in Figure 5.5b This is a fairly naive parallel breadth-first search implementation where multiple threads work through the *todo* queue and perform term rewriting steps in parallel to determine the states reachable from the current state. With protection sets, two threads are already sufficient to outperform all sequential implementations, and we achieve a speed up of 12 for 16 threads, and a speed up of 16 for 32 threads. Reference counting is clearly a less viable option, most likely due to heavy contention on the atomic reference counters due to the amount of sharing.

5.5 Conclusion

We present a formal specification of a *thread-safe* term library, *i.e.*, a term library that can be used by multiple threads in concurrently, and several behavioural properties. We identify the need for a *readers-writer* lock implementation, *i.e.*, multiple threads can read a resource simultaneously or a single writer can obtain *exclusive* access, that is efficient for read heavy workloads. To this end we develop a new *busy-forbidden* protocol that only writes one atomic variable twice that is typically in the local cache and reads another variable that is rarely updated to obtain read access. For this new protocol we also develop a formal specification and model check the corresponding implementation model for key properties on finite instances up to seven threads. Furthermore, we have also model checked the specification of the complete term library for several finite instances. Finally, we have implement this term library in the mCRL2 toolset and show that the state space generation scales well up to 16 threads on a machine with 32 physical cores. Furthermore, for the micro-benchmarks we observe that it scales well for the shared term case and generally we observe that protection sets scale far better than reference counting as a garbage collection protection mechanism. Therefore, we can conclude that for heavy use of maximal sharing we can achieve

fairly linear speed up, but also with the insight that maximal sharing comes at a large cost for parallelism.

We consider the verification of the busy-forbidden protocol, and eventually the term library itself, for any number of threads important future work. Furthermore, there are several orthogonal features present in the actual implementation on top of the busy-forbidden protocol that have not been formally verified. For example, in practice we required a mechanism to allow nested entries into the shared section, which we have implemented by means of a counter. Another possible extension would be to allow *upgrading* a shared section into an exclusive section, which is also a feature commonly found in readers-writer locks. To this end, it would also be interesting to compare various implementations of readers-writer locks to each other practically. This would be a large undertaking since the implementations of some protocols are unavailable and furthermore we have found that the pseudocode sometimes contained insufficient information to be implemented reliably.

Other practical improvements would be to introduce more sophisticated thread-safe hash table implementations for comparison to our naive thread-safe hash table implementation. Similarly, introducing concurrent garbage collection techniques could be implemented to further improve the scalability of the approach. Finally, we have only parallelised the state space exploration process, but it would also be interesting to parallelise other expensive steps used in verification, for example the state space minimisation or the refinement checking algorithms presented in Chapter 2.

Chapter 6

Conclusions and Future Work

In this dissertation we have studied improvements to various algorithms that are used for the verification of concurrent systems. This has led to promising practical improvements to these algorithms, but also showed many open challenges in verifying actual practical systems efficiently. All presented algorithms have been implemented and are available in the open source toolset mCRL2 [23], including the new tools *lpscleave* and *lpscombine* to perform the cleave procedure presented in Chapter 3. This means that they can be used and also form the basis for further (practical) improvements. In the remainder of this chapter we will reflect on the proposed research questions and the work that has been carried out as part of this dissertation.

First, in Chapter 2 we have concerned ourselves with the following research question.

RQ1 Can we resolve the issues pertaining to the correctness and efficiency of the antichain-based refinement algorithm presented in [137]?

We have shown that all three algorithms perform suboptimally when implemented using a depth-first search strategy and poorly when implemented using a breadth-first search strategy. Furthermore, all three algorithms violate the claimed antichain property. In Chapter 2 we proposed new algorithms for which we have shown correctness and which utilise proper antichains. Our experiments indicate significant performance improvements for deciding trace refinement, stable failures refinement and a performance of deciding failures-divergences refinement that is comparable to deciding stable failures refinement. We also show that preprocessing using divergence-preserving branching bisimulation offers substantial performance benefits. Therefore,

we can answer this research question positively. The resulting algorithms are currently used as the backbone in the commercial F-MDE toolset Dezyne; see also [8].

We proposed several ideas of future work related to these algorithms, for example comparison to the state-of-the-art or between more classical approaches to deciding refinement. Also comparing different state space minimisation techniques, or applying the antichain based technique to decide other refinement algorithms such as fair testing [119]. In addition to this, with the introduction of our thread-safe term library it would be interesting to see if the refinement algorithms can be parallelised, for example by exploring different parts of the implementation model concurrently. Similar algorithms have already been implemented in FDR4.

As a next step, we concerned ourselves with effectively applying compositional minimisation to a monolithic process by decomposing it into a number of parallel components to efficiently obtain a reduced state space, in order to answer the following research question.

RQ2 Can compositional minimisation be applied to a monolithic process by decomposing it into multiple monolithic processes?

In Chapter 3 we have presented a decomposition technique, a cleave, that can be applied to any monolithic process as an LPE and have shown that it is always a valid decomposition, which answers this research question positively. Furthermore, we have shown that state invariants can be used to improve the effectiveness of the decomposition. However, we have also found out that deriving an effective parameter partitioning can be difficult. Many experiments had to be performed in order to show that the technique *can* be effective. This means that in its current form it is not yet suited for a practical application in most general cases, and finding the most effective decomposition using heuristics is important future work.

On a more positive note, recent work has shown that the developed *lpscleave* and *lpscombine* tools can successfully be used for compositional minimisation for specifications where there are a number of clearly separated components, which was not (easily) achievable in the mCRL2 toolset without these tools. Furthermore, additional work has been carried out to apply a cleave to the typically more useful abstraction based on (divergence-preserving) branching bisimulation minimisation [56]. Overall, this shows promising results towards the improvements of these algorithms and the development of robust compositional minimisation within the mCRL2 toolset. Furthermore, since the introduction of efficient symbolic model checking techniques within the mCRL2 toolset, it could also be interesting to determine how two symbolically represented state spaces could be efficiently composed.

Next, in Chapter 4 we have concerned ourselves with the following research

question.

RQ3 Can on-the-fly solving be applied effectively to symbolic parity games?

To this end, we have developed the theory to reason about on-the-fly solving of parity games, independent of the strategy that is used to explore games. In addition to that, we have introduced the notion of *safe* vertices, shown their correctness, proven an optimality result, and we have studied partial solvers and shown that these can be made to run without determining the safe vertices first; which can be useful for on-the-fly solving. Finally, we have demonstrated the practical purpose of our method and observed that solitaire winning cycle detection with safe attractors is almost always beneficial with minimal overhead, but also that more powerful partial solvers can be useful. Therefore, we can conclude that this research question has a positive answer.

We have hinted at several improvements to the on-the-fly algorithms such as choosing the most efficient solver, or practical improvements to the heuristics and procedures used in the symbolic solving algorithms. Overall, the introduction of symbolic model checking to the mCRL2 toolset has resulted in being able to verify state space far larger than was possible with explicit state model checking. This includes automatically verifying that the first player has a winning strategy for the connect four game played on the full 7x6 board, which resulted in a parity game of about 10^{16} vertices. On-the-fly solving was only one of the features available in the explicit state verification tools that has now been transferred to the symbolic setting.

An important feature missing from the symbolic verification tools is the ability to generate an effective counter example for failing verification, or alternatively a witness for succeeding verification. Applying the technique presented in [138] to symbolic parity games is theoretically sound, but is not practically feasible in the way that it is currently implemented. However, this is crucial when trying to understand why the modelled behaviour is wrong, or why the modal formula is ill-formulated, and therefore important future work. Furthermore, the techniques employed in symbolic verification within the mCRL2 toolset are rather primitive and state-of-the-art symbolic verification tools, such as LTSmin, implement far more sophisticated heuristics and algorithms that should be studied and incorporated in order to further scale up the models that can be verified.

Finally, another important aspect for scaling is the ability utilise multi-core processors efficiently, for which we have posed the following research question.

RQ4 Can a *thread-safe* term library to store and manipulate terms be developed that achieves *linear speedup* compared to an efficient sequential implementation?

In Chapter 5 we present a formal specification of a *thread-safe* term library, *i.e.*, a term library that can be used by multiple threads in parallel. We have designed and verified an efficient *readers-writer* lock implementation, *i.e.*, multiple threads can share a resource or a writer can obtain *exclusive* access, that is efficient for read heavy workloads called the *busy-forbidden* protocol. Finally, we implement this term library in the mCRL2 toolset and show that the state space generation scales well for multiple threads, with the observation that protection sets scale better than reference counting as protection mechanism. Unfortunately, in general we must concede that no linear speed up could be achieved for cases where threads operate on different terms. However, for the case where terms are heavily shared among each other as well as among the threads a speed up of 10 was achieved with 32 threads on a processor with 32 cores.

Therefore, the conclusion would be that for cases where terms are not heavily shared, the overhead of maximal sharing (*i.e.*, hash consing) in a thread-safe context is generally undesirable with our proposed implementation. However, we have also demonstrated that for our practical use case of state space exploration it scales well up to 16 threads on a machine with 32 physical cores, due to the amount of sharing that takes place. Overall, the verification of the busy-forbidden protocol and the term library revealed several scalability issues. For example the checking equivalence of the busy-forbidden protocol implementation and specification took more than a week of computation power. Here, investigating the implementation of symbolic equivalence checking, or minimisation, algorithms could help deal with this problem, for example the ones described in [82] or [40]. Similarly, the thread-safe term library was only verified for a small number of threads, terms and addresses due to the cost of verification efforts. Therefore, it remains the case that scalability of formal verification algorithms should be an active area of research for the foreseeable future, and most likely a combination of techniques, including the ones described in this dissertation, are necessary in order to overcome the challenge of formally verifying large scale software systems.

Appendices

Appendix A

Antichain-based Refinement Checking Proofs

This appendix contains detailed proofs for several auxiliary lemmas used in Chapter 2.

A.1 Proof of Lemma 2.2.4

Lemma 2.2.4. Let $\mathscr{L} = (S, \iota, \to)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \to')$. For all sequences $\rho \in Act^*$ and states $U \in S'$ such that $\iota' \overset{\rho}{\to\!\!\!\!\to}' U$, it holds that $\iota \overset{\rho}{\Longrightarrow} s$ for all $s \in [\![U]\!]_{\mathscr{L}}$.

Proof. We use induction on the length of sequences in Act^* to prove the statement.

Base case. First, $\iota' \overset{\varepsilon}{\to\!\!\!\!\to}' U$ iff $U = \iota'$ by definition. The state ι' is equal to $\{s \mid \iota \overset{\varepsilon}{\Longrightarrow} s\}$ as defined in the normalisation. Hence, for every state $s \in [\![\iota']\!]_{\mathscr{L}}$ we have $\iota \overset{\varepsilon}{\Longrightarrow} s$.

Inductive case. Pick any sequence $\rho \in Act^*$ of length i and suppose that the statement holds for all sequences in Act^* of length i. Take an arbitrary state $V \in S'$ and action $a \in Act$ such that $\iota' \overset{\rho\, a}{\to\!\!\!\!\to}' V$. Then there is a state $U \in S'$ such that $\iota' \overset{\rho}{\to\!\!\!\!\to}' U$ and $U \overset{a}{\to}' V$. By definition of normalisation there is a transition $U \overset{a}{\to}' V$ if and only if $V = \{t \in S \mid \exists s \in U : s \overset{a}{\Longrightarrow} t\}$. So for all states $t \in [\![V]\!]_{\mathscr{L}}$ there is a state $s \in [\![U]\!]_{\mathscr{L}}$ such that $s \overset{a}{\Longrightarrow} t$. By the induction hypothesis it holds that for all $s \in [\![U]\!]_{\mathscr{L}}$ there is a weak transition $\iota \overset{\rho}{\Longrightarrow} s$. But then we may conclude that $\iota \overset{\rho\, a}{\Longrightarrow} t$ for all $t \in [\![V]\!]_{\mathscr{L}}$. □

A.2 Proof of Lemma 2.2.5

Lemma 2.2.5. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and for all states $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$, there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$.

Proof. We proceed using an induction on the length of sequences in Act^*.

Base case. Let $s \in S$ and suppose $\iota \stackrel{\varepsilon}{\Longrightarrow} s$. Then $s \in [\![\iota']\!]_{\mathscr{L}}$, since ι' is defined as $\{s \in S \mid \iota \stackrel{\varepsilon}{\Longrightarrow} s\}$. Moreover, we trivially have $\iota' \stackrel{\varepsilon}{\twoheadrightarrow}' \iota'$.

Inductive step. Pick any sequence $\rho \in Act^*$ of length i and suppose that the statement holds for all sequences in Act^* of length i. Take an arbitrary state $t \in S$ and action $a \in Act$ such that $\iota \stackrel{\rho a}{\Longrightarrow} t$. Then there is a state $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$ and $s \stackrel{a}{\Longrightarrow} t$. Fix such a state $s \in S$. From the induction hypothesis it then follows that there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$. Fix this state $U \in S'$. Let V be equal to $\{t \in S \mid \exists u \in U : u \stackrel{a}{\Longrightarrow} t\}$; then by definition, $U \stackrel{a}{\rightarrow}' V$. It follows that $\iota' \stackrel{\rho a}{\twoheadrightarrow}' V$. Finally, $t \in [\![V]\!]_{\mathscr{L}}$ follows from $s \stackrel{a}{\Longrightarrow} t$ and $s \in [\![U]\!]_{\mathscr{L}}$. □

A.3 Proof of Lemma 2.2.6

Lemma 2.2.6. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ it holds that $\rho \notin \mathsf{weaktraces}(\mathscr{L})$ if and only if $\iota' \stackrel{\rho}{\twoheadrightarrow}' \emptyset$.

Proof.

$\Longrightarrow$) We use induction on the length of the sequences in Act^*.

Base case. The implication holds vacuously since $\varepsilon \in \mathsf{weaktraces}(\mathscr{L})$.

Inductive step. Pick a sequence $\rho \in Act^*$ of length i and suppose that the implication holds for all sequences of length i. Assume an arbitrary action $a \in Act$ such that $\rho\, a \notin \mathsf{weaktraces}(\mathscr{L})$. From $\rho\, a \notin \mathsf{weaktraces}(\mathscr{L})$ it follows that there is no state $t \in S$ such that $\iota \stackrel{\rho a}{\Longrightarrow} t$. We distinguish two cases:

- Case $\rho \notin \mathsf{weaktraces}(\mathscr{L})$. From the induction hypothesis we obtain $\iota' \stackrel{\rho}{\rightarrow} \rightarrow' \emptyset$ and $\emptyset \stackrel{a}{\rightarrow}' \emptyset$ by definition. We may therefore also conclude $\iota' \stackrel{\rho a}{\twoheadrightarrow}' \emptyset$.
- Case $\rho \in \mathsf{weaktraces}(\mathscr{L})$. Then there is a state $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$. Since $\mathscr{L}'$ is deterministic there is a unique state $U \in S'$ such that both $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$ by Lemmas 2.2.5 and 2.1.4. For all states $u \in S$ satisfying $\iota \stackrel{\rho}{\Longrightarrow} u$ (which exist as $\rho \in \mathsf{weaktraces}(\mathscr{L})$) there cannot be a state $t \in S$ such that $u \stackrel{a}{\Longrightarrow} t$ by the observation that $\rho\, a \notin \mathsf{weaktraces}(\mathscr{L})$. Therefore, $U \stackrel{a}{\rightarrow}' \emptyset$ and thus also $\iota' \stackrel{\rho a}{\twoheadrightarrow}' \emptyset$.

$\Longleftarrow$) Suppose $\iota' \overset{\rho}{\twoheadrightarrow}' \emptyset$. Towards a contradiction, assume that $\rho \in \mathsf{weaktraces}(\mathscr{L})$. Then there is a state $s \in S$ such that $\iota \overset{\rho}{\Longrightarrow} s$. By Lemma 2.2.5, there must be some $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \overset{\rho}{\twoheadrightarrow}' U$. Since $\mathscr{L}'$ is deterministic, by Lemma 2.1.4, we obtain that this state U must be such that $U = \emptyset$. □

A.4 Proof of Lemma 2.2.18

Lemma 2.2.18. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ such that either $\rho \notin \mathsf{divergences}(\mathscr{L})$ or $\rho \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$ and for all states $s \in S$ such that $\iota \overset{\rho}{\Longrightarrow} s$ there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \overset{\rho}{\twoheadrightarrow}' U$.

Proof. Proof by induction on the length of sequences that are not divergences, or minimal divergences.

Base case. The empty trace ε satisfies $\varepsilon \notin \mathsf{divergences}(\mathscr{L})$ or $\varepsilon \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$ by definition. Hence, we must show that for all states $s \in S$ satisfying $\iota \overset{\varepsilon}{\Longrightarrow} s$ there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \overset{\varepsilon}{\twoheadrightarrow}' U$. We know that if $\iota \overset{\varepsilon}{\Longrightarrow} s$ then $s \in [\![\iota']\!]_{\mathscr{L}}$, because ι' is defined as $\{s \in S \mid \iota \overset{\varepsilon}{\Longrightarrow} s\}$ in the normalisation. Finally, we also know that $\iota' \overset{\varepsilon}{\twoheadrightarrow}' \iota'$.

Inductive step. Suppose that the statement holds for all sequences of length i that are either not divergences or minimal divergences of $\mathscr{L}$. Pick a sequence $\rho \in Act^*$ of length i, an arbitrary state $t \in S$ and action $a \in Act$ such that $\iota \overset{\rho a}{\Longrightarrow} t$ and $\rho a \notin \mathsf{divergences}(\mathscr{L})$ or $\rho a \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$. Note that whenever $\rho a \notin \mathsf{divergences}(\mathscr{L})$ or $\rho a \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$ then $\rho \notin \mathsf{divergences}(\mathscr{L})$. From $\iota \overset{\rho a}{\Longrightarrow} t$ it follows that there is a state $s \in S$ such that $\iota \overset{\rho}{\Longrightarrow} s$ and $s \overset{a}{\Longrightarrow} t$. By our induction hypothesis it then follows that there is a state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \overset{\rho}{\twoheadrightarrow}' U$. For all states $u \in [\![U]\!]_{\mathscr{L}}$ it holds that $\iota \overset{\rho}{\Longrightarrow} u$ by Lemma 2.2.17, so by definition of divergences it must be that $u\Uparrow$ does not hold and hence $[\![U]\!]_{\mathscr{L}}\Uparrow$ does not hold. Let V be equal to $\{v \in S \mid \exists u \in U : u \overset{a}{\Longrightarrow} v\}$ such that $U \overset{a}{\rightarrow}' V$ by definition of the normalisation. It follows that $\iota' \overset{\rho a}{\twoheadrightarrow}' V$. Finally, $t \in [\![V]\!]_{\mathscr{L}}$ follows from $s \overset{a}{\Longrightarrow} t$ and $s \in [\![U]\!]_{\mathscr{L}}$. □

A.5 Proof of Lemma 2.2.19

Lemma 2.2.19. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ and states $U \in S'$ it holds that if $\iota' \overset{\rho}{\twoheadrightarrow}' U$ and not $[\![U]\!]_{\mathscr{L}}\Uparrow$ then $\rho \notin \mathsf{divergences}(\mathscr{L})$.

Proof. Let $\rho \in Act^*$ and $U \in S'$ be such that $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$ and not $[\![U]\!]_{\mathscr{L}}\Uparrow$. Towards a contradiction, assume that $\rho \in \mathsf{divergences}(\mathscr{L})$. We distinguish two cases:

- $\rho \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$. Let $t \in S$ be such that $\iota \stackrel{\rho}{\Longrightarrow} t$ and $t\Uparrow$. Note that due to the determinism of $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L})$ and Lemma 2.2.18, for all $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$, we have $s \in U$. Hence also $t \in U$. But $t\Uparrow$ then implies $[\![U]\!]_{\mathscr{L}}\Uparrow$. Contradiction.
- $\rho \notin \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$. Then $\rho = \rho'\rho''$ for some $\rho' \in \mathsf{divergences}_{\mathsf{min}}(\mathscr{L})$. Let $t \in S$ be such that $\iota \stackrel{\rho'}{\Longrightarrow} t$ and $t\Uparrow$. Then, by Lemma 2.2.18, there must be some $V \in S'$ such that $\iota' \stackrel{\rho'}{\twoheadrightarrow}' V$ and $t \in V$. Let V be such. Since $t\Uparrow$ and $t \in V$, V has no outgoing transitions in $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L})$. In particular, we cannot have $V \stackrel{\rho''}{\twoheadrightarrow}' U$, and because $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L})$ is deterministic, we also cannot have $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$. Contradiction. □

A.6 Proof of Lemma 2.2.20

Lemma 2.2.20. Let $\mathscr{L} = (S, \iota, \rightarrow)$ be an LTS and let $\mathsf{norm}_{\mathsf{fdr}}(\mathscr{L}) = (S', \iota', \rightarrow')$. For all sequences $\rho \in Act^*$ it holds that $\rho \notin (\mathsf{divergences}(\mathscr{L}) \cup \mathsf{weaktraces}(\mathscr{L}))$ if and only if $\iota' \stackrel{\rho}{\twoheadrightarrow}' \emptyset$.

Proof.

$\Longrightarrow$) Proof by induction on the length of sequences in Act^*.
Base case. The implication holds vacuously since $\varepsilon \in \mathsf{weaktraces}(\mathscr{L})$.
Inductive step. Suppose that the statement holds for all sequences Act^* of length i. Pick a sequence $\rho \in Act^*$ of length i. Take an arbitrary action $a \in Act$ such that $\rho\, a \notin (\mathsf{divergences}(\mathscr{L}) \cup \mathsf{weaktraces}(\mathscr{L}))$. From $\rho\, a \notin \mathsf{weaktraces}(\mathscr{L})$ it follows that there is no state $t \in S$ such that $\iota \stackrel{\rho\, a}{\Longrightarrow} t$. From $\rho\, a \notin \mathsf{divergences}(\mathscr{L})$ it follows that $\rho \notin \mathsf{divergences}(\mathscr{L})$. Now, there are two cases to distinguish:

- Case $\rho \notin \mathsf{weaktraces}(\mathscr{L})$. From the induction hypothesis we obtain $\iota' \stackrel{\rho}{\rightarrow} \rightarrow' \emptyset$ and $\emptyset \stackrel{a}{\rightarrow}' \emptyset$ by definition. Thus $\iota' \stackrel{\rho\, a}{\twoheadrightarrow}' \emptyset$.
- Case $\rho \in \mathsf{weaktraces}(\mathscr{L})$. There is a state $s \in S$ such that $\iota \stackrel{\rho}{\Longrightarrow} s$. Since $\mathscr{L}'$ is deterministic there is a unique state $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \stackrel{\rho}{\twoheadrightarrow}' U$ by Lemma 2.2.18 and 2.1.4. We may furthermore conclude that $[\![U]\!]_{\mathscr{L}}\Uparrow$ does not hold. Because $\rho\, a \notin \mathsf{weaktraces}(\mathscr{L})$, no state $u \in S$ for which $\iota \stackrel{\rho}{\Longrightarrow} u$ satisfies $u \stackrel{a}{\Longrightarrow} t$, for any state t. Therefore, by definition, $U \stackrel{a}{\rightarrow}' \emptyset$, and thus $\iota' \stackrel{\rho\, a}{\twoheadrightarrow}' \emptyset$.

$\Longleftarrow$) Suppose $\iota' \overset{\rho}{\twoheadrightarrow}' \emptyset$. From the observation that $[\![\emptyset]\!]_{\mathscr{L}} \Uparrow$ does not hold and Lemma 2.2.19 it follows that $\rho \notin \mathsf{divergences}(\mathscr{L})$. Towards a contradiction, assume that $\rho \in \mathsf{weaktraces}(\mathscr{L})$. Then there is a state $s \in S$ such that $\iota \overset{\rho}{\Longrightarrow} s$. By Lemma 2.2.20, there must be some $U \in S'$ such that $s \in [\![U]\!]_{\mathscr{L}}$ and $\iota' \overset{\rho}{\twoheadrightarrow}' U$. Since $\mathscr{L}'$ is deterministic, by Lemma 2.1.4, we obtain that this state U must be such that $U = \emptyset$. Contradiction. $\square$

Appendix B

Threadsafe Term Library Formalisation

This appendix contains a detailed explanation of the mCRL2 process specifications and modal mu-calculus formulae used to verify the thread-safe term library described in Chapter 5

B.1 mCRL2 Specifications for the Busy-Forbidden Protocol

In this section we give the formal mCRL2 specifications of the implementation and the external behaviour (or specification) of the busy forbidden protocol that are used to perform the model checking and equivalence checking. Entering the shared section is specified in Table B.1 and leaving it in Table B.2. Note that we use actions to model the assignments to variables, for example $store_p(Busy(p), \mathsf{true}, p)$ corresponds to the assignment of true to *p.busy* in the implementation pseudocode. The process algebra has no global variables and we use an additional process and actions to read from and write to these variables. For the atomic flags we introduce a struct F that is defined below to declare a busy and a forbidden flag per thread.

$$\texttt{sort}\ F = \texttt{struct}\ Busy(P) \mid Forbidden(P)$$

Table B.3 shows the behaviour of the *Busy* and *Forbidden* flags for every thread and the *mutex* variable. We model the 'while' construction in the pseudocode by recursion

$$
\begin{array}{l}
\mathit{EnterShared}(p:P) = \\
\quad \texttt{enter\_shared\_call}(p)\ . \\
\quad \mathit{TryBothFlags}(p)\ . \\
\quad \texttt{enter\_shared\_return}(p) \\
\\
\mathit{TryBothFlags}(p:P) = \\
\quad \mathit{store}_p(\mathit{Busy}(p), \mathsf{true}, p)\ .\ (\\
\quad\quad \mathit{load}_p(\mathit{Forbidden}(p),\ \mathsf{true},\ p)\ . \\
\quad\quad \mathit{store}_p(\mathit{Busy}(p), \mathsf{false}, p)\ .\ \mathit{improbable}\ .\ \mathit{TryBothFlags}(p) \\
\quad + \ \mathit{load}_p(\mathit{Forbidden}(p), \mathsf{false}, p) \\
\quad)
\end{array}
$$

Table B.1: mCRL2 specification for the implementation of `enter_shared`.

$$
\begin{array}{l}
\mathit{LeaveShared}(p:P) = \\
\quad \texttt{leave\_shared\_call}(p)\ . \\
\quad \mathit{store}_p(\mathit{Busy}(p), \mathsf{false}, p)\ . \\
\quad \texttt{leave\_shared\_return}(p)
\end{array}
$$

Table B.2: mCRL2 specification for the implementation of `leave_shared`.

and have added the *improbable* action to ensure equivalence modulo divergence-preserving branching bisimulation. Similarly, entering the exclusive section is specified in Table B.4 and leaving it in Table B.5. Observe that we use a typewriter font (for example `enter_shared_call`) to indicate visible actions and a italics front (for example store_p) to indicate internal actions that will be hidden for divergence-preserving branching bisimulation reductions. Here, we use a set *forbidden* (and for leaving *allowed*) to keep track of the threads whose forbidden flag has already been set to true.

Table B.6 shows the specification for the behaviour of a thread. Each thread repeatedly tries to (non-deterministic) enter and leave either a shared or exclusive section. Finally, Table B.7 contains the complete mCRL2 specification of the various processes in a parallel composition and the necessary communication to deal with the atomic flags and mutex.

B.1.1 Modal Formulas

In this section we explain the modal formulas corresponding to the informal properties listed in Section 5.2.3. The formula for the property 1 is shown in Table B.9 and states that when a thread enters the exclusive section, no other thread may enter that

$$
\begin{array}{l}
Flags(flags : F \to Bool) = \\
\sum_{f:F,p:P} . (\\
\quad \sum_{b:Bool} . store_f(f,b,p) \,.\, Flag(flags[f \mapsto b]) \\
\quad + \; load_f(f,flags(f),p) \,.\, Flag(flags) \\
\quad) \\
\\
Mutex(locked : Bool) = \\
\sum_{p:P} . (\\
\quad locked \\
\quad \to lock_m(p) \,.\, Mutex(\mathsf{true}) \\
\quad \diamond \; unlock_m(p) \,.\, Mutex(\mathsf{false}) \\
\quad)
\end{array}
$$

Table B.3: mCRL2 specifications for the atomic flags and the mutex.

$$
\begin{array}{l}
EnterExclusive(p : P) = \\
\quad \texttt{enter\_exclusive\_call}(p) \\
\quad lock_p(p) \,. \\
\quad SetAllForbiddenFlags(p,\, \emptyset) \,. \\
\quad \texttt{enter\_exclusive\_return}(p) \\
\\
SetAllForbiddenFlags(p : P,\, forbidden : Set(P)) = \\
\quad (\forall_{p':P} . p \in forbidden) \\
\quad \to internal \\
\quad \diamond \; \sum_{p':P} . store_p(Forbidden(p'), \mathsf{true}, p) \,.\, (\\
\qquad load_p(Busy(p'), \mathsf{false}, p) \,. \\
\qquad SetAllForbiddenFlags(p,\, forbidden \cup \{p'\}) \\
\qquad + \; load_p(Busy(p'), \mathsf{true}, p) \,. \\
\qquad store_p(Forbidden(p'), \mathsf{false}, p) \,.\, improbable \,. \\
\qquad SetAllForbiddenFlags(p,\, forbidden \setminus \{p'\}) \\
\qquad + \; store_p(Forbidden(p'), \mathsf{false}, p) \,.\, improbable \,. \\
\qquad SetAllForbiddenFlags(p,\, forbidden \setminus \{p'\}) \\
\qquad)
\end{array}
$$

Table B.4: mCRL2 specification for the `enter_exclusive` function shown in Table 5.2.

$$
\begin{array}{l}
\mathit{LeaveExclusive}(p : P) = \\
\quad \texttt{leave\_exclusive\_call}(p)\ . \\
\quad \mathit{AllowAllThreads}(p, \emptyset)\ . \\
\quad \mathit{unlock}_p(p)\ . \\
\quad \texttt{leave\_exclusive\_return}(p) \\
\\
\mathit{AllowAllThreads}(p : P,\ \mathit{allowed} : \mathit{Set}(P)) = \\
\quad (\forall_{q:P}.q \in \mathit{allowed}) \\
\quad \rightarrow \mathit{internal} \\
\quad \diamond\ \textstyle\sum_{p':P} .(\\
\quad\quad \mathit{store}_p(\mathit{Forbidden}(p'), \mathsf{false}, p)\ . \\
\quad\quad \mathit{AllowAllThreads}(p,\ \mathit{allowed} \cup \{p'\}) \\
\quad + \ \mathit{store}_p(\mathit{Forbidden}(p'), \mathsf{true}, p)\ .\ \mathit{improbable} \\
\quad\quad \mathit{AllowAllThreads}(p,\ \mathit{allowed} \setminus \{p'\}) \\
\quad\quad)
\end{array}
$$

Table B.5: mCRL2 specification for the `leave_exclusive` function shown in Table 5.2.

$$
\begin{array}{l}
\mathit{Thread}(p : P) = \\
\quad \mathit{EnterShared}(p)\ . \\
\quad \mathit{LeaveShared}(p)\ . \\
\quad \mathit{Thread}(p) \\
+ \ \mathit{EnterExclusive}(p)\ . \\
\quad \mathit{LeaveExclusive}(p)\ . \\
\quad \mathit{Thread}(p)
\end{array}
$$

Table B.6: mCRl2 specification for a thread p interacting with the protocol.

allow({
store, *load*,
lock, *unlock*,
internal, improbable,
`enter_shared_call`, `enter_shared_return`,
`leave_shared_call`, `leave_shared_return`,
`enter_exclusive_call`, `enter_exclusive_return`,
`leave_exclusive_call`, `leave_exclusive_return`
}, **comm**({
$store_f|store_p \rightarrow store$,
$load_f|load_p \rightarrow load$,
$lock_m|lock_p \rightarrow lock$,
$unlock_m|unlock_p \rightarrow unlock$
},
$Thread(p_1)$ ||
⋮
$Thread(p_{|P|})$ ||
$Flags(\lambda f{:}F.\mathsf{false})$ ||
$Mutex(\mathsf{false})$
)
)

Table B.7: mCRL2 specification for the busy-forbidden protocol.

section till it leaves the section. The formulas for properties 3 and 4 are presented in Tables B.8 and B.10 and use data parameters to count the number of threads in the exclusive section or in any section respectively. Furthermore, these formulas and the formula for property 5 in Table B.11 utilise the following construction that was presented in Equation 5.3.1 of Section 5.3.1 to state that some action a must be able to occur. Observe that under strong fairness assumptions this means that this action actually will occur. Note that these are two subformulas with identical structure for shared and exclusive sections respectively.

$$\nu X.\mu Y.([\overline{a \cup b}]Y \wedge [b]X \wedge \langle \mathsf{true}^* \,.\, a \rangle \mathsf{true})$$

Finally, the property 6 presented in Table B.12 uses boolean parameters to keep track of whether any thread is in the shared or exclusive sections respectively. This is more efficient than keeping track of the exact amount of threads.

$$
\begin{array}{l}
\nu X(n_{exclusive} : Nat = 0). \\
\quad [\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p)]X(n_{exclusive}+1) \\
\wedge\ [\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p)]X(n_{exclusive}-1) \\
\wedge\ [\\
\quad \overline{(\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p))} \\
\quad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p))} \\
\quad]\, X(n_{exclusive}) \\
\wedge\ \forall_{p:P}.[\texttt{enter\_shared\_call}(p)] \\
\quad \nu Y(n'_{exclusive} : Nat = n_{exclusive}).\mu Z(n''_{exclusive} : Nat = n'_{exclusive}).\ (\\
\quad\quad [\\
\quad\quad \overline{\texttt{enter\_shared\_return}(p)} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p'))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p'))} \\
\quad\quad \cap\ \overline{improbable} \\
\quad\quad]\ (\\
\quad\quad\quad ((n''_{exclusive} \approx 0) \implies Z(n''_{exclusive})) \\
\quad\quad \wedge\ ((n''_{exclusive} > 0) \implies Y(n''_{exclusive})) \\
\quad\quad\quad) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')]Y(n''_{exclusive}) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')]Y(n''_{exclusive}+1) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p')]Y(n''_{exclusive}-1) \\
\quad \wedge\ [improbable]Y(n''_{exclusive}) \\
\quad \wedge\ \langle \mathsf{true}^*\,.\,\texttt{enter\_shared\_return}(p)\rangle \mathsf{true} \\
\quad\quad) \\
\end{array}
$$

Table B.8: Modal formula for property 3: “When a thread requests to enter the shared section, it will be granted access within a bounded number of steps, unless there is another thread in the exclusive section”.

$$
\begin{array}{l}
[\mathsf{true}^*] \\
[\exists_{p\in P} : \texttt{enter\_exclusive\_return}(p)] \\
[\overline{\exists_{p\in P} : \texttt{leave\_exclusive\_call}(p)}^{\,*}] \\
[\exists_{p\in P} : \texttt{enter\_exclusive\_return}(p)] \\
\mathsf{false}
\end{array}
$$

Table B.9: Modal formula for property 1: “There should never be more than one thread present in the exclusive section”.

$$
\begin{array}{l}
\nu X(n_{blocking} : Nat = 0). \\
\quad [\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p)]X(n_{blocking}+1) \\
\quad \wedge\ [\exists_{p:P}.\ \texttt{enter\_shared\_call}(p)]X(n_{blocking}+1) \\
\quad \wedge\ [\exists_{p:P}.\ \texttt{leave\_shared\_return}(p)]X(n_{blocking}-1) \\
\quad \wedge\ [\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p)]X(n_{blocking}-1) \\
\quad \wedge\ [\\
\qquad \overline{(\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p))} \\
\qquad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p))} \\
\qquad \cap\ \overline{(\exists_{p:P}.\ \texttt{enter\_shared\_call}(p))} \\
\qquad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_shared\_return}(p))} \\
\qquad]\ X(n_{exclusive}) \\
\quad \wedge\ \forall_{p:P}.[\texttt{enter\_exclusive\_call}(p)] \\
\qquad \nu Y(n'_{blocking} : Nat = n_{blocking}).\mu Z(n''_{blocking} : Nat = n'_{blocking}).\ (\\
\qquad\quad [\\
\qquad\quad \overline{\texttt{enter\_exclusive\_return}(p)} \\
\qquad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p))} \\
\qquad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_shared\_return}(p))} \\
\qquad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\qquad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p'))} \\
\qquad\quad \cap\ \overline{\mathit{improbable}} \\
\qquad\quad]\ (\\
\qquad\qquad ((n''_{blocking} \approx 0) \implies Z(n''_{blocking})) \\
\qquad\quad \wedge\ ((n''_{blocking} > 0) \implies Y(n''_{blocking})) \\
\qquad\qquad) \\
\qquad \wedge\ [\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')]Y(n''_{blocking}+1) \\
\qquad \wedge\ [\exists_{p':P}.\ \texttt{leave\_shared\_return}(p')]Y(n''_{blocking}-1) \\
\qquad \wedge\ [\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')]Y(n''_{blocking}+1) \\
\qquad \wedge\ [\exists_{p':P}.\ (p' \not\approx p) \wedge \texttt{enter\_exclusive\_return}(p')]Y(n''_{blocking}-1) \\
\qquad \wedge\ [\mathit{improbable}]Y(n''_{exclusive}) \\
\qquad \wedge\ \langle \mathsf{true}^* \,.\, \texttt{enter\_exclusive\_return}(p) \rangle \mathsf{true} \\
\qquad\quad) \\
\end{array}
$$

Table B.10: Modal formula for property 4: “When a thread requests to enter the exclusive section, it will be granted access within a bounded number of steps, unless there is another thread in the shared or in the exclusive section”.

$$
\begin{array}{l}
[\mathsf{true}^*]\ \forall_{p:P}.(\\
\quad [\mathtt{leave\_shared\_call}(p)]\nu X.\mu Y.(\\
\quad\quad [\ \overline{\mathtt{leave\_shared\_return}(p)} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \mathtt{enter\_exclusive\_call}(p'))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \mathtt{enter\_shared\_call}(p'))} \\
\quad\quad \cap\ \overline{\mathit{improbable}} \\
\quad\quad]\ Y \\
\quad\quad \wedge\ [\\
\quad\quad\quad (\exists_{p':P}.\ \mathtt{enter\_exclusive\_call}(p')) \\
\quad\quad\quad \cup\ (\exists_{p':P}.\ \mathtt{enter\_shared\_call}(p')) \\
\quad\quad\quad \cup\ (\mathit{improbable}) \\
\quad\quad\quad]\ X \\
\quad\quad \wedge\ \langle \mathsf{true}^*\,.\,\mathtt{leave\_shared\_return}(p)\rangle \mathsf{true} \\
\quad\quad) \\
\quad \wedge\ [\mathtt{leave\_exclusive\_call}(p)]\nu X.\mu Y.(\\
\quad\quad [\ \overline{\mathtt{leave\_exclusive\_return}(p)} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \mathtt{enter\_exclusive\_call}(p'))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \mathtt{enter\_shared\_call}(p'))} \\
\quad\quad \cap\ \overline{\mathit{improbable}} \\
\quad\quad]\ Y \\
\quad\quad \wedge\ [\\
\quad\quad\quad (\exists_{p':P}.\ \mathtt{enter\_exclusive\_call}(p')) \\
\quad\quad\quad \cup\ (\exists_{p':P}.\ \mathtt{enter\_shared\_call}(p')) \\
\quad\quad\quad \cup\ (\mathit{improbable}) \\
\quad\quad\quad]\ X \\
\quad\quad \wedge\ \langle \mathsf{true}^*\,.\,\mathtt{leave\_exclusive\_return}(p)\rangle \mathsf{true} \\
\quad\quad) \\
\quad)
\end{array}
$$

Table B.11: Modal formula for property 5: "When a thread requests to leave the exclusive/shared section, it will leave it within a bounded number of steps".

$$
\begin{array}{l}
\forall_{p:P}.\nu X(b_{shared} : Bool = \mathsf{false},\ b_{exclusive} : Bool = \mathsf{false}). \\
\quad [\texttt{enter\_shared\_call}(p)]X(\mathsf{true}, b_{exclusive}) \\
\wedge\ [\texttt{leave\_shared\_return}(p)]X(\mathsf{false}, b_{exclusive}) \\
\wedge\ [\texttt{enter\_exclusive\_call}(p)]X(b_{shared}, \mathsf{true}) \\
\wedge\ [\texttt{leave\_exclusive\_return}(p)]X(b_{shared}, \mathsf{false}) \\
\wedge\ [\\
\quad \overline{\texttt{enter\_shared\_call}(p)} \\
\quad \cap\ \overline{\texttt{leave\_shared\_return}(p)} \\
\quad \cap\ \overline{\texttt{enter\_exclusive\_call}(p)} \\
\quad \cap\ \overline{\texttt{leave\_exclusive\_return}(p)} \\
\quad]\, X(n_{shared}, n_{exclusive}) \\
\wedge\ ((\neg n_{shared} \wedge \neg n_{exclusive}) \implies (\\
\quad \langle \texttt{enter\_exclusive\_call(p)} \rangle \mathsf{true} \\
\quad \wedge\ \langle \texttt{enter\_shared\_call(p)} \rangle \mathsf{true} \\
\quad))
\end{array}
$$

Table B.12: Modal formula for property 6: "A thread not in the exclusive or shared section can instantly start to enter the exclusive or shared section".

B.2 mCRL2 Specifications for the Term Library

In this section we give the formal mCRL2 specifications of the implementation and the external behaviour (or specification) of the term library forbidden protocol that are used to perform the model checking. Creating a term is specified in Table B.13 and destroying a term in Table B.16. In this model, the set P corresponds to the set containing all threads, T to the set containing all terms and A to the set containing all memory addresses. The set $A_\perp = A \cup \{\perp\}$ with $\perp \notin A$ contains the extra element $\perp$ meaning no address or a NULL pointer. To ensure finiteness and reduce the complexity of the model, the set T only contains a finite amount of constants, *i.e.*, terms of arity zero.

First of all, we introduce processes *EnterShared*, *LeaveShared*, *EnterExclusive* and *LeaveExclusive* to interact with the busy-forbidden specification *BF* specified in Table 5.3. To distinguish between the term library and the protocol all actions such as `enter_shared_call` are split into action $\texttt{enter\_shared\_call}_{bf}$ for the protocol and $\texttt{enter\_shared\_call}_{p}$ for the term library. Finally, we have the process *MainMemory* to model the main memory by keeping track of *used* memory addresses, the process *HashTable* which model a hash table as an associative array and process *ReferenceCounter* to track a reference counter for every address (or term). Destroying a term is specified in Table B.16, which uses the same other processes as the creation

function. Again there are two separate processes to model the behaviour of the while loop.

The specification in Table B.18 models the behaviour of each thread. Each thread repeatedly tries to either create a term it does not yet know, which means that it is not destroyed, or destroys one it does. Finally, Table B.17 shows the complete specification including the communication between various processes used to model the thread-safe term library. The **comm** and **allow** operators specify the communication between the various components.

B.2.1 Requirements as Modal Formulas

To verify the model of the thread-safe term library we again specify a number of modal formulas for the properties described in Section 5.1.2. The modal formula for property 1 specified in Table B.19 uses mapping a from addresses to terms and the finite set *owners* containing all threads that own/protect term t as data parameters. If at any point in time a `create(t)` returns a different address than the current address, then the term must not be owned by any thread. The modal formula in Table B.20 for property 2 uses the same constructs to check whether terms on the same address are also equivalent.

The formula for property 3 shown in Table B.21 uses a boolean parameter *busy* to keep track of whether the thread p is creating (or destroying) a term. Furthermore, the parameter *known* is a finite set containing all terms that thread p knows. If at any point in time *busy* is false, then the process must be able to start destroying any term in *known* and start creating any term not currently in *known*. Finally, for property 4 the formula shown in Table B.22 uses again the construction which (under fairness) indicates that term creating (and destroying) will finish within a finite number of steps. Note that these are two subformulas with identical structure for creation and destruction respectively.

$$
\begin{array}{l}
\mathit{Create}(p:P,\ t:T,\ lm:T \to A_\perp) = \\
\quad \texttt{create\_call}(p,\ t)\ . \\
\quad \mathit{EnterShared}(p)\ . \\
\quad \mathit{Create}_2(p,\ t,\ lm) \\
\\
\mathit{Create}_2(p:P,\ t:T,\ lm:T \to A_\perp) = \\
\sum_{a:A_\perp} . (\\
\quad \mathit{contains}_p(t,a,p)\ . \\
\quad (a \approx \perp) \\
\quad \to \sum_{a':A} . \ (\\
\qquad \mathit{construct\_term}_p(t\ ,a'\ ,p) . (\\
\qquad\quad \mathit{insert}_p(t\ ,a'\ ,\mathsf{true}\ ,p)\ . \\
\qquad\quad \mathit{Create}_3(p,\ t,\ lm,\ a') \\
\qquad + \ \mathit{insert}_p(t,\ a',\ \mathsf{false},\ p)\ . \\
\qquad\quad \mathit{destruct\_term}_p(t,\ a',\ p)\ . \\
\qquad\quad \mathit{Create}_2(p,\ t,\ lm) \\
\qquad\quad)) \\
\quad \diamond\ \ \mathit{Create}_3(p,\ t,\ lm,\ a) \\
\quad) \\
\\
\mathit{Create}_3(p:P,t:T,\ lm:T \to A_\perp,\ a:A) = \\
\quad \mathit{protect}_p(t,\ a,\ p)\ . \\
\quad \mathit{LeaveShared}(p)\ . \\
\quad \texttt{create\_return}(p,\ t,\ a)\ . \\
\quad \mathit{Thread}(p,\ lm[t \mapsto a])
\end{array}
$$

Table B.13: mCRL2 specification for the create function shown in Table 5.1.

$$
\begin{aligned}
&\mathit{EnterShared}(p : P) = \\
&\qquad \mathit{enter\_shared\_call}_p(p) \,. \\
&\qquad \mathit{enter\_shared\_return}_p(p) \\
&\\
&\mathit{LeaveShared}(p : P) = \\
&\qquad \mathit{leave\_shared\_call}_p(p) \,. \\
&\qquad \mathit{leave\_shared\_return}_p(p) \\
&\\
&\mathit{EnterExclusive}(p : P) = \\
&\qquad \mathit{enter\_exclusive\_call}_p(p) \,. \\
&\qquad \mathit{enter\_exclusive\_return}_p(p) \\
&\\
&\mathit{LeaveExclusive}(p : P) = \\
&\qquad \mathit{leave\_exclusive\_call}_p(p) \,. \\
&\qquad \mathit{leave\_exclusive\_return}_p(p)
\end{aligned}
$$

Table B.14: mCRL2 processes used to communicate with the busy-forbidden protocol.

$$
\begin{array}{l}
MainMemory(used : FSet(A)) = \\
\sum_{p:P,t:T,a:A} \cdot (\\
\quad (a \notin used) \\
\quad \rightarrow construct\_term_{mm}(t,a,p) \,.\, MainMemory(used \cup \{a\}) \\
\quad \diamond \;\; destruct\_term_{mm}(t,a,p) \,.\, MainMemory(used \setminus \{a\}) \\
\quad) \\
\\
HashTable(m : T \rightarrow A_{\perp}) = \\
\sum_{t:T,p:P} \cdot (\\
\quad contains_{ht}(t, m(e), p) \,.\, HashTable(m) \\
\quad + \;\; \sum_{a:A} \cdot (m(e) \approx \perp) \\
\quad \rightarrow insert_{ht}(t,a,\mathsf{true},p) \,.\, HashTable(m[e \mapsto a) \\
\quad \diamond \;\; insert_{ht}(t,a,\mathsf{false},p) \,.\, HashTable(m) \\
\quad + \;\; delete_{ht}(t,p) \,.\, HashTable(m[e \mapsto \perp]) \\
\quad) \\
\\
ReferenceCounter(counter : A \rightarrow Nat) = \\
\quad \sum_{t:T,p:P} \,.\, protect_{rc}(t,a,p) \;. \\
\quad ReferenceCounter(counter[a \mapsto counter(a)+1] \\
\quad + \;\; \sum_{t:T,p:P} \,.\, unprotect_{rc}(t,a,p) \;. \\
\quad ReferenceCounter(counter[a \mapsto counter(a)-1] \\
\quad + \;\; \sum_{t:T,p:P} \,.\, protected_{rc}(t,a,counter(a) \not\approx 0, p) \;. \\
\quad ReferenceCounter(counter)
\end{array}
$$

Table B.15: mCRL2 specifications of the main memory, hash table and reference counters used in the term library specification.

```
Destroy(p : P, t : T, lm : T → A⊥) =
    destroy_call(p, t) .
    unprotect_p(t, lm(t), p) . (
      skip
    + skip . GC(p) ) .
    destroy_return(p) .
    Thread(p, lm[t ↦ ⊥])

GC(p : P) =
    EnterExclusive(p) .
    GC_2(p, ∅)

GC_2(p : P, checked : FSet(T)) =
    (∀_{t:T} . t ∈ checked)
    → LeaveExclusive(p)
    ◇ ∑_{t:T} . (t ∉ checked) → (
        contains_p(t, ⊥, p) .
        GC_2(p, checked ∪ {t})
      + ∑_{a:A} . contains_p(t, a, p) . (
          protected_p(a, true, p) .
          GC_2(p, checked ∪ {t})
        + protected_p(a, false, p) .
          destruct_term_p(t, a, p) .
          delete_p(t, p) .
          GC_2(p, checked ∪ {t})
          )
        )
```

Table B.16: mCRL2 specification for the destroy function shown in Table 5.1.

$$
\begin{aligned}
&\textbf{allow}(\{\\
&\quad \mathit{construct\_term}, \mathit{destruct\_term},\\
&\quad \mathit{contains}, \mathit{insert}, \mathit{delete},\\
&\quad \mathit{protect}, \mathit{unprotect}, \mathit{protected},\\
&\quad \mathit{skip},\\
&\quad \mathit{improbable},\\
&\quad \mathit{enter\_shared\_call}, \mathit{enter\_shared\_return},\\
&\quad \mathit{leave\_shared\_call}, \mathit{leave\_shared\_return},\\
&\quad \mathit{enter\_exclusive\_call}, \mathit{enter\_exclusive\_return},\\
&\quad \mathit{leave\_exclusive\_call}, \mathit{leave\_exclusive\_return},\\
&\quad \texttt{create\_call}, \texttt{create\_return},\\
&\quad \texttt{destroy\_call}, \texttt{destroy\_return}\\
&\quad \},\textbf{comm}(\{\\
&\qquad \mathit{construct\_term}_{mm}|\mathit{construct\_term}_{p} \rightarrow \mathit{construct\_term},\\
&\qquad \mathit{destruct\_term}_{mm}|\mathit{destruct\_term}_{p} \rightarrow \mathit{destruct\_term},\\
&\qquad \mathit{contains}_{ht}|\mathit{contains}_{p} \rightarrow \mathit{contains},\\
&\qquad \mathit{insert}_{ht}|\mathit{insert}_{p} \rightarrow \mathit{insert},\\
&\qquad \mathit{delete}_{ht}|\mathit{delete}_{p} \rightarrow \mathit{delete},\\
&\qquad \mathit{protect}_{rc}|\mathit{protect}_{p} \rightarrow \mathit{protect},\\
&\qquad \mathit{unprotect}_{rc}|\mathit{unprotect}_{p} \rightarrow \mathit{unprotect},\\
&\qquad \mathit{protected}_{rc}|\mathit{protected}_{p} \rightarrow \mathit{protected},\\
&\qquad \mathit{enter\_shared\_call}_{bf}|\mathit{enter\_shared\_call}_{p} \rightarrow \mathit{enter\_shared\_call},\\
&\qquad \mathit{enter\_shared\_return}_{bf}|\mathit{enter\_shared\_return}_{p} \rightarrow \mathit{enter\_shared\_return},\\
&\qquad \mathit{leave\_shared\_call}_{bf}|\mathit{leave\_shared\_call}_{p} \rightarrow \mathit{leave\_shared\_call},\\
&\qquad \mathit{leave\_shared\_return}_{bf}|\mathit{leave\_shared\_return}_{p} \rightarrow \mathit{leave\_shared\_return},\\
&\qquad \mathit{enter\_exclusive\_call}_{bf}|\mathit{enter\_exclusive\_call}_{p} \rightarrow \mathit{enter\_exclusive\_call},\\
&\qquad \mathit{enter\_exclusive\_return}_{bf}|\mathit{enter\_exclusive\_return}_{p} \rightarrow\\
&\mathit{enter\_exclusive\_return},\\
&\qquad \mathit{leave\_exclusive\_call}_{bf}|\mathit{leave\_exclusive\_call}_{p} \rightarrow \mathit{leave\_exclusive\_call},\\
&\qquad \mathit{leave\_exclusive\_return}_{bf}|\mathit{leave\_exclusive\_return}_{p} \rightarrow \mathit{leave\_exclusive\_return}\\
&\qquad \},\\
&\qquad \mathit{Thread}(p_1) \mid\mid\\
&\qquad\quad \vdots\\
&\qquad \mathit{Thread}(p_{|P|}) \mid\mid\\
&\qquad \mathit{MainMemory}(\emptyset) \mid\mid\\
&\qquad \mathit{HashTable}(\lambda t{:}T.\bot) \mid\mid\\
&\qquad \mathit{ReferenceCounter}(\lambda a{:}A.\ 0) \mid\mid\\
&\qquad \mathit{BF}(\lambda p{:}P.\mathit{Free})\\
&\qquad)\\
&\quad)
\end{aligned}
$$

Table B.17: mCRL2 specification for the thread-safe term library.

$$
\begin{array}{l}
Thread(p:P,\ lm:T \to A_\bot) = \\
\quad (\sum_{t:T}.(lm(t) \approx \bot) \to Create(p,t,lm)) \\
+\ \ (\sum_{t:T}.(lm(t) \not\approx \bot) \to Destroy(p,t,lm))
\end{array}
$$

Table B.18: mCRL2 specification of a thread p interacting with the term library.

$$
\begin{array}{l}
\forall_{t:T}.\nu X(a:A_\bot = \bot,\ owners:FSet(P) = \emptyset). \\
\quad (\forall_{p:P,a':A}. \\
\quad\quad [\texttt{create\_return}(p,t,a')]\ (\\
\quad\quad\quad X(a', owners \cup \{p\}) \\
\quad\quad\quad \wedge\ (a \not\approx a' \implies owners \approx \emptyset) \\
\quad\quad\quad) \\
\quad\quad) \\
\quad \wedge\ (\forall p:P.\ [\texttt{destroy\_call}(p,t)]X(a, owners \setminus \{p\})) \\
\quad \wedge\ [\\
\quad\quad \overline{\exists_{p:P,a':A}.\texttt{create\_return}(p,t,a')} \\
\quad\quad \cap\ \overline{\exists_{p:P}.\texttt{destroy\_call}(p,t)} \\
\quad\quad]\ X(a, owners)
\end{array}
$$

Table B.19: Formulation of property 1: "A term and all its subterms remain in existence at exactly the same address, with unchanged function symbol and arguments, as long as it is not destroyed".

$$
\begin{array}{l}
\forall_{a:A,t_1:T}.\nu X(t:T = t_1, owners:FSet(P) = \emptyset). \\
\quad (\forall_{p:P,t_2:T}. \\
\quad\quad [\texttt{create\_return}(p,t_2,a)]\ (\\
\quad\quad\quad X(t_2, owners \cup \{p\}) \\
\quad\quad\quad \wedge\ (t \not\approx t_2 \implies owners \approx \emptyset) \\
\quad\quad\quad) \\
\quad\quad) \\
\quad \wedge\ (\forall_{p:P}.\ [\texttt{destroy\_call}(p,t)]X(t, owners \setminus \{p\})) \\
\quad \wedge\ [\\
\quad\quad \overline{\exists_{p:P,t':T}.\texttt{create\_return}(p,t',a)} \\
\quad\quad \cap\ \overline{\exists_{p:P}.\texttt{destroy\_call}(p,t)} \\
\quad\quad]\ X(t, owners)
\end{array}
$$

Table B.20: Modal formula for property 2: "Two accessible terms t_1 and t_2 always have the same non-null address iff they are equal".

$$
\begin{array}{l}
\forall_{p:P}.\ \nu X(busy : Bool = \mathsf{false}, known : FSet(T) = \emptyset). \\
\quad (\neg busy) \rightarrow (\\
\qquad (\forall_{t:T}.(t \notin known) \implies [\tau^*]\langle \tau^*.\texttt{create\_call}(p,t)\rangle \mathsf{true}) \\
\quad \wedge\ (\forall_{t:T}.(t \in known) \implies [\tau^*]\langle \tau^*.\texttt{destroy\_call}(p,t)\rangle \mathsf{true}) \\
\qquad) \\
\wedge\ [\\
\qquad (\exists_{t:T}.\texttt{create\_call}(p,t)) \\
\quad \cup\ (\exists_{t:T}.\texttt{destroy\_call}(p,t)) \\
\quad]\, X(\mathsf{true}, known) \\
\wedge\ (\forall_{t:T}.\ [\exists_{a:A}.\texttt{create\_return}(p,t,a)]X(\mathsf{false}, known \cup \{t\})) \\
\wedge\ (\forall_{t:T}.\ [\texttt{destroy\_return}(p,t)]X(\mathsf{false}, known \setminus \{t\})) \\
\wedge\ [\\
\qquad \overline{(\exists_{t:T}.\texttt{create\_call}(p,t))} \\
\quad \cap\ \overline{(\exists_{t:T}.\texttt{destroy\_call}(p,t))} \\
\quad \cap\ \overline{(\exists_{t:T,a:A}.\texttt{create\_return}(p,t,a))} \\
\quad \cap\ \overline{(\exists_{t:T}.\texttt{destroy\_return}(p,t))} \\
\quad]\, X(busy, known)
\end{array}
$$

Table B.21: Modal formula for property 3: “Any thread that is not busy creating or destroying a term, can always initiate the creation of a new term or the destruction of any term that this thread has access to”.

$$
\begin{array}{l}
([\mathsf{true}^*]\forall_{p:P,t:T}.[\mathtt{create\_call}(p,t)]\ \nu X_c.\ \mu Y_c.(\\
\quad \forall_{p':P}.(p \not\approx p') \implies \\
\qquad [\\
\qquad\quad (\exists_{t':T}.\ \mathtt{create\_call}(p',t')) \\
\qquad \cup\ (\exists_{t':T}.\ \mathtt{destroy\_call}(p',t')) \\
\qquad \cup\ \mathit{improbable} \\
\qquad]\ X_c \\
\quad \wedge\ [\\
\qquad \overline{(\exists_{a:A}.\ \mathtt{create\_return}(p,t,a))} \\
\qquad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \mathtt{create\_call}(p',t')))} \\
\qquad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \mathtt{destroy\_call}(p',t')))} \\
\qquad \cap\ \overline{\mathit{improbable}} \\
\qquad]\ Y_c \\
\quad \wedge\ \langle \mathsf{true}^*.\exists_{a:A}.\ \mathtt{create\_return}(p,t,a)\rangle \mathsf{true} \\
\qquad)) \\
\wedge \\
([\mathsf{true}^*]\forall_{p:P,t:T}.[\mathtt{destroy\_call}(p,t)]\ \nu X_d.\ \mu Y_d.(\\
\quad \forall_{p':P}.(p \not\approx p') \implies \\
\qquad [\\
\qquad\quad (\exists_{t':T}.\ \mathtt{create\_call}(p',t')) \\
\qquad \cup\ (\exists_{t':T}.\ \mathtt{destroy\_call}(p',t')) \\
\qquad \cup\ \mathit{improbable} \\
\qquad]\ X_d \\
\quad \wedge\ [\\
\qquad \overline{\mathtt{destroy\_return}(p,t)} \\
\qquad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \mathtt{create\_call}(p',t')))} \\
\qquad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \mathtt{destroy\_call}(p',t')))} \\
\qquad \cap\ \overline{\mathit{improbable}} \\
\qquad]\ Y_d \\
\quad \wedge\ \langle \mathsf{true}^*.\mathtt{destroy\_return}(p,t)\rangle \mathsf{true} \\
\qquad))
\end{array}
$$

Table B.22: Modal formula(s) for property 4: "Any thread that started creating a term or destroying a term, will eventually successfully finish this task provided there is enough memory to store one more term than those that are accessible. But it is required that other threads behave fairly, in the sense that they will not continually create and destroy terms or stall other threads by busy waiting".

B.3 Experimental Results as Tables

In this section we show the exact values for the values that are shown in the plots in Figures 5.4 and 5.5.

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	0.03	0.11	0.22	0.14	0.26	0.34	0.28	0.29	0.39	0.51	0.53
parallel protection set	0.03	0.07	0.07	0.20	0.15	0.28	0.20	0.17	0.32	0.34	0.32
sequential reference counter	0.02										
sequential protection set	0.02										
original aterm library	0.01										
parallel java	0.26	0.46	0.68	1.32	1.25	1.20	1.51	1.41	1.36	1.48	1.51
std::shared_mutex	0.03	0.12	0.18	0.32	0.24	0.22	0.38	0.47	0.47	0.55	0.50
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	0.51	0.59	0.50	0.54	0.49	0.53	0.54	0.48	0.5	0.48	0.52
parallel protection set	0.35	0.39	0.32	0.39	0.33	0.41	0.39	0.38	0.38	0.37	0.39
parallel java	1.40	1.51	1.44	1.43	1.38	1.67	1.77	1.87	1.77	1.78	1.85
std::shared_mutex	0.58	0.62	0.63	0.67	0.72	0.74	0.76	0.79	0.83	0.83	0.88
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	0.50	0.53	0.50	0.49	0.53	0.51	0.51	0.49	0.48	0.48	
parallel protection set	0.43	0.37	0.39	0.41	0.39	0.39	0.42	0.44	0.39	0.41	
parallel java	1.68	1.95	1.68	1.82	1.81	1.65	1.94	1.94	2.09	1.86	
std::shared_mutex	0.91	0.95	0.96	0.99	0.98	1.04	1.02	1.10	1.11	1.16	

Table B.23: Wall-clock time in seconds for creating new terms (shared) shown in Figure 5.4a.

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	0.03	0.03	0.03	0.04	0.03	0.04	0.05	0.07	0.06	0.06	0.06
parallel protection set	0.03	0.05	0.03	0.04	0.03	0.03	0.04	0.05	0.05	0.05	0.06
sequential reference counter	0.02										
sequential protection set	0.02										
original aterm library	0.01										
parallel java	0.26	0.25	0.28	0.26	0.28	0.28	0.28	0.35	0.33	0.35	0.32
std::shared_mutex	0.03	0.04	0.04	0.05	0.10	0.04	0.04	0.09	0.09	0.09	0.09
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	0.06	0.05	0.05	0.07	0.06	0.06	0.05	0.06	0.07	0.06	0.06
parallel protection set	0.05	0.06	0.06	0.05	0.06	0.07	0.06	0.05	0.06	0.06	0.06
parallel java	0.33	0.35	0.32	0.34	0.33	0.34	0.33	0.35	0.33	0.34	0.34
std::shared_mutex	0.09	0.10	0.10	0.10	0.10	0.09	0.11	0.09	0.12	0.11	0.13
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	0.07	0.06	0.05	0.06	0.06	0.07	0.07	0.07	0.06	0.06	
parallel protection set	0.06	0.06	0.05	0.07	0.06	0.06	0.05	0.05	0.05	0.06	
parallel java	0.37	0.34	0.35	0.35	0.37	0.35	0.34	0.34	0.35	0.34	
std::shared_mutex	0.13	0.11	0.14	0.09	0.15	0.18	0.15	0.13	0.16	0.13	

Table B.24: Wall-clock time in seconds for creating new terms (distinct) shown in Figure 5.4b.

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	9.01	35.1	11.5	19.0	9.81	10.6	6.40	6.00	4.90	4.85	3.88
parallel protection set	4.07	2.51	1.66	1.39	1.07	0.96	0.81	0.75	0.67	0.59	0.53
sequential reference counter	4.01										
sequential protection set	3.65										
original aterm library	4.57										
parallel java	104	136	106	103	91.7	84.2	75.5	71.1	64.5	61.6	57.5
std::shared_mutex	6.51	14.2	15.1	15.1	18.7	20.7	22.0	33.3	28.1	25.4	24.5
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	3.51	3.10	2.86	2.77	2.66	2.45	2.61	2.43	2.33	2.27	2.17
parallel protection set	0.49	0.46	0.43	0.41	0.40	0.39	0.37	0.36	0.35	0.32	0.31
parallel java	54.3	51.9	49.4	48.7	46.5	47.7	47.9	48.1	48.6	47.7	46.6
std::shared_mutex	22.8	22.7	23.1	22.6	22.5	22.9	23.2	23.7	24.7	24.1	24.3
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.16	2.10	2.08	2.04	2.03	1.97	1.82	1.87	1.81	1.81	
parallel protection set	0.32	0.31	0.29	0.3	0.28	0.27	0.29	0.28	0.28	0.29	
parallel java	45.9	45.4	45.8	44.9	44.8	42.4	42.4	42.9	42.1	42.1	
std::shared_mutex	24.7	24.4	25.0	25.1	25.4	25.5	26.0	26.7	27.5	28.3	

Table B.25: Wall-clock time for creating existing terms (shared).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	9.02	5.15	3.95	3.05	2.64	2.25	2.38	2.41	2.42	2.49	2.24
parallel protection set	3.96	2.66	2.58	2.44	2.27	1.76	1.92	1.86	1.95	1.91	1.79
sequential reference counter	4.02										
sequential protection set	3.71										
original aterm library	4.58										
parallel java	106	212	218	227	260	266	274	276	295	272	287
std::shared_mutex	6.46	13.8	15.5	15.7	18.3	18.5	24.6	33.2	26.9	25.0	23.7
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.22	2.27	2.26	2.20	2.39	2.55	2.63	2.58	2.72	2.67	2.67
parallel protection set	1.78	1.76	1.77	1.75	1.81	1.82	1.97	2.05	2.04	2.07	2.07
parallel java	275	287	296	2912	281	286	280	284	294	292	314
std::shared_mutex	22.7	22.4	22.9	22.6	22.2	22.7	23.2	23.8	24.7	24.1	24.1
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.69	2.69	2.77	2.76	2.8	2.77	2.82	2.84	2.86	2.92	
parallel protection set	2.07	2.10	2.15	2.05	2.12	2.11	2.17	2.22	2.27	2.22	
parallel java	311	308	315	316	324	330	339	332	343	352	
std::shared_mutex	24.4	24.4	25.3	25.3	25.9	25.7	26.3	27.1	27.8	28.7	

Table B.26: Wall-clock time for creating existing terms (distinct).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	15.7	8.63	5.93	4.60	3.87	3.41	3.00	2.79	2.50	2.45	2.21
parallel protection set	16.7	8.90	6.07	4.66	3.93	3.37	3.01	2.80	2.55	2.41	2.34
sequential reference counter	16.8										
sequential protection set	18.2										
original aterm library	16.4										
parallel java	34.6	34.5	36.0	36.7	36.1	33.6	30.9	28.4	26.4	25.0	22.9
std::shared_mutex	16.2	8.71	5.95	4.54	3.86	3.34	3.01	2.74	2.53	2.40	2.29
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.17	2.21	2.23	2.32	2.24	2.14	2.30	2.21	2.11	2.21	2.09
parallel protection set	2.28	2.21	2.30	2.35	2.21	2.26	2.35	2.25	2.33	2.28	2.21
parallel java	17.8	20.6	17.7	19.1	22.3	19.5	19.6	20.4	21.9	21.6	21.5
std::shared_mutex	2.25	2.33	2.28	2.24	2.21	2.22	2.29	2.15	2.24	2.04	2.24
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.17	2.18	2.13	2.07	2.06	2.09	2.06	2.05	2.13	2.10	
parallel protection set	2.26	2.15	2.12	2.16	2.13	2.07	2.09	2.16	2.03	2.03	
parallel java	17.6	19.2	18.4	20.6	22.7	20.8	18.5	22.5	23.6	19.4	
std::shared_mutex	2.24	2.12	2.18	2.05	2.05	2.2	2.16	2.17	2.02	2.07	

Table B.27: Wall-clock time for traversing terms (shared).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	18.4	9.61	6.41	4.93	4.10	3.56	3.14	2.88	2.63	2.51	2.34
parallel protection set	17.0	8.80	6.03	4.59	3.85	3.39	3.00	2.78	2.56	2.40	2.28
sequential reference counter	15.9										
sequential protection set	18.3										
original aterm library	17.4										
parallel java	34.5	34.2	35.8	37.0	35.5	33.8	30.1	28.3	27.1	23.4	21.9
std::shared_mutex	16.5	8.59	5.98	4.63	3.99	3.47	3.07	2.88	2.60	2.49	2.43
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.36	2.33	2.33	2.28	2.24	2.19	2.26	2.22	2.16	2.20	2.11
parallel protection set	2.31	2.38	2.28	2.31	2.20	2.21	2.21	2.13	2.21	2.16	2.18
parallel java	21.1	17.5	20.9	17.3	18.7	20.8	23.0	18.4	21.6	23.0	22.8
std::shared_mutex	2.56	2.52	2.50	2.41	2.32	2.25	2.4	2.37	2.25	2.34	2.39
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.27	2.16	2.16	2.13	2.10	2.34	2.13	2.06	2.16	2.05	
parallel protection set	2.27	2.15	2.16	2.17	2.12	2.08	2.11	2.10	2.06	2.04	
parallel java	18.3	22.2	22.3	22.5	18.7	21.7	22.2	19.3	22.8	19.5	
std::shared_mutex	2.18	2.30	2.27	2.34	2.14	2.28	2.08	2.10	2.38	2.26	

Table B.28: Wall-clock time for traversing terms (distinct).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	61.0	87.6	90.1	83.1	67.0	57.4	56.2	53.3	50.0	46.5	43.4
parallel protection set	62.8	31.7	21.5	16.5	13.4	11.2	9.79	8.67	7.78	7.15	6.54
sequential reference counter	60.0										
sequential protection set	52.9										
original aterm library	40.2										
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	43.3	41.4	38.8	38.1	37.3	35.5	35.2	35.2	33.9	33.2	32.5
parallel protection set	6.07	5.68	5.37	5.06	4.85	4.83	4.72	4.67	4.60	4.56	4.49
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	31.7	31.3	30.8	29.4	28.7	28.1	28.0	27.9	27.0	26.6	
parallel protection set	4.44	4.37	4.32	4.23	4.17	4.15	4.11	4.07	4.02	3.99	

Table B.29: Wall-clock time for state space exploration.

#Threads	1	2	3	4	5	6	7	8
parallel reference counter	10.0	10.4	4.42	4.08	3.22	2.74	2.38	2.22
parallel protection set	5.68	3.09	2.36	1.60	1.52	1.30	1.14	1.02
sequential reference counter	4.61							
sequential protection set	4.20							
original aterm library	4.83							
parallel java	130	73.0	50.4	40.5	32.1	29.5	28.5	29.2
std::shared_mutex	10.3	38.1	42.1	41.0	40.7	41.7	42.7	46.2

Table B.30: Wall-clock time for creating existing terms (shared, Intel).

Bibliography

[1] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. "When Simulation Meets Antichains". In: *TACAS*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. LNCS. Springer, 2010, pp. 158–174. DOI: 10.1007/978-3-642-12002-2_14.

[2] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Vol. 18. Cambridge tracts in theoretical computer science. Cambridge University Press, 1990. ISBN: 978-0-521-40043-5.

[3] I. Beer, S. Ben-David, and A. Landver. "On-the-Fly Model Checking of RCTL Formulas". In: *CAV*. Ed. by A. Hu and M. Vardi. Vol. 1427. LNCS. Springer, 1998, pp. 184–194. DOI: 10.1007/BFb0028744.

[4] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Solving parity games via priority promotion". In: *Formal Methods Syst. Des.* 52.2 (2018), pp. 193–226. DOI: 10.1007/s10703-018-0315-1.

[5] J. Bergstra and P. Klint. "The TOOLBUS Coordination Architecture". In: *COORDINATION*. Ed. by P. Ciancarini and C. Hankin. Vol. 1061. LNCS. Springer, 1996, pp. 75–88. DOI: 10.1007/3-540-61052-9_40.

[6] J. A. Bergstra and J. W. Klop. "ACP$_\tau$: A Universal Axiom System for Process Specification". In: *Algebraic Methods: Theory, Tools and Applications*. Ed. by M. Wirsing and J. A. Bergstra. Vol. 394. LNCS. Springer, 1987, pp. 447–463. DOI: 10.1007/BFb0015048.

[7] J. A. Bergstra, J. W. Klop, and E. Olderog. "Failures without chaos: a new process semantics for fair abstraction". In: *Formal Description of Programming Concepts 1986*. Ed. by M. Wirsing. North-Holland, 1987, pp. 77–104.

[8] R. van Beusekom, J. F. Groote, P. F. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, and T. A. C. Willemse. "Formalising the Dezyne Modelling Language in mCRL2". In: *FMICS-AVoCS*. Ed. by L. Petrucci, C. Seceleanu, and A. Cavalcanti. Vol. 10471. LNCS. Springer, 2017, pp. 217–233. DOI: 10.1007/978-3-319-67113-0_14.

[9] M. Bezem and J. F. Groote. "Invariants in Process Algebra with Data". In: *CONCUR*. Ed. by B. Jonsson and J. Parrow. Vol. 836. LNCS. Springer, 1994, pp. 401–416. DOI: 10.1007/978-3-540-48654-1_30.

[10] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *IFM 2017*. Ed. by N. Polikarpova and S. A. Schneider. Vol. 10510. LNCS. Springer, 2017, pp. 102–110. DOI: 10.1007/978-3-319-66845-1_7.

[11] S. Blom, J. F. Groote, S. Mauw, and A. Serebrenik. "Analysing the BKE-security Protocol with μCRL". In: *Electron. Notes Theor. Comput. Sci.* 139.1 (2005), pp. 49–90. DOI: 10.1016/j.entcs.2005.09.005.

[12] S. Blom, J. van de Pol, and M. Weber. "LTSmin: Distributed and Symbolic Reachability". In: *CAV*. Ed. by T. Touili, B. Cook, and P. B. Jackson. Vol. 6174. LNCS. Springer, 2010, pp. 354–359. DOI: 10.1007/978-3-642-14295-6_31.

[13] A. Boulgakov, T. Gibson-Robinson, and A. W. Roscoe. "Computing maximal weak and other bisimulations". In: *Formal Aspects Comput.* 28.3 (2016), pp. 381–407. DOI: 10.1007/s00165-016-0366-2.

[14] P. Bouvier, H. Garavel, and H. P. de León. "Automatic Decomposition of Petri Nets into Automata Networks - A Synthetic Account". In: *PETRI NETS 2020*. Ed. by R. Janicki, N. Sidorova, and T. Chatain. Vol. 12152. LNCS. Springer, 2020, pp. 3–23. DOI: 10.1007/978-3-030-51831-8_1.

[15] M. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. "Efficient annotated terms". In: *Softw. Pract. Exp.* 30.3 (2000), pp. 259–291. DOI: 10.1002/(SICI)1097-024X(200003)30:3\<259::AID-SPE298\>3.0.CO;2-Y.

[16] M. van den Brand and P. Klint. "ATerms for manipulation and exchange of structured data: It's all about sharing". In: *Inf. Softw. Technol.* 49.1 (2007), pp. 55–64. DOI: 10.1016/j.infsof.2006.08.009.

[17] M. van den Brand, P. Moreau, and J. J. Vinju. "Generator of efficient strongly typed abstract syntax trees in Java". In: *IEE Proc. Softw.* 152.2 (2005), pp. 70–78. DOI: 10.1049/ip-sen:20041181.

[18] E. Brinksma, R. Langerak, and P. Broekroelofs. “Functionality Decomposition by Compositional Correctness Preserving Transformation”. In: *CAV*. Ed. by C. Courcoubetis. Vol. 697. LNCS. Springer, 1993, pp. 371–384. DOI: 10.1007/3-540-56922-7_31.

[19] S. D. Brookes and A. W. Roscoe. “An Improved Failures Model for Communicating Processes”. In: *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*. Ed. by S. D. Brookes, A. W. Roscoe, and G. Winskel. Vol. 197. LNCS. Springer, 1984, pp. 281–305. DOI: 10.1007/3-540-15670-4_14.

[20] G. Bruns and P. Godefroid. “Model Checking Partial State Spaces with 3-Valued Temporal Logics”. In: *CAV 1999*. Ed. by N. Halbwachs and D. A. Peled. Vol. 1633. LNCS. Springer, 1999, pp. 274–287. DOI: 10.1007/3-540-48683-6_25.

[21] V. Bruyère, G. A. Pérez, J. Raskin, and C. Tamines. “Partial Solvers for Generalized Parity Games”. In: *RP*. Ed. by E. Filiot, R. M. Jungers, and I. Potapov. Vol. 11674. LNCS. Springer, 2019, pp. 63–78. DOI: 10.1007/978-3-030-30806-3_6.

[22] R. E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. In: *ACM Comput. Surv.* 24.3 (1992), pp. 293–318. DOI: 10.1145/136035.136043.

[23] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. “The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability”. In: *TACAS*. Ed. by T. Vojnar and L. Zhang. Vol. 11428. LNCS. Springer, 2019, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2.

[24] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. “NUMA-aware reader-writer locks”. In: *ACM SIGPLAN 2013*. Ed. by A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc. ACM, 2013, pp. 157–166. DOI: 10.1145/2442516.2442532.

[25] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. “Deciding parity games in quasipolynomial time”. In: *STOC*. Ed. by H. Hatami, P. McKenzie, and V. King. ACM, 2017, pp. 252–263. DOI: 10.1145/3055399.3055409.

[26] T. Chen, B. Ploeger, J. van de Pol, and T. A. C. Willemse. “Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems”. In: *CONCUR*. Ed. by L. Caires and V. T. Vasconcelos. Vol. 4703. LNCS. Springer, 2007, pp. 120–135. DOI: 10.1007/978-3-540-74407-8_9.

[27] K. Cheng and A. S. Krishnakumar. “Automatic Functional Test Generation Using the Extended Finite State Machine Model”. In: *Proceedings of the 30th Design Automation Conference*. Ed. by A. E. Dunlop. ACM Press, 1993, pp. 86–91. DOI: 10.1145/157485.164585.

[28] S. Cheung and J. Kramer. “Context Constraints for Compositional Reachability Analysis”. In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (1996), pp. 334–377. DOI: 10.1145/235321.235323.

[29] G. Ciardo, R. M. Marmorstein, and R. Siminiceanu. “The saturation algorithm for symbolic state-space exploration”. In: *Int. J. Softw. Tools Technol. Transf.* 8.1 (2006), pp. 4–25. DOI: 10.1007/s10009-005-0188-7.

[30] R. Cleaveland, M. Klein, and B. Steffen. “Faster Model Checking for the Modal Mu-Calculus”. In: *CAV*. Ed. by G. von Bochmann and D. K. Probst. Vol. 663. LNCS. Springer, 1992, pp. 410–422. DOI: 10.1007/3-540-56496-9_32.

[31] P. Courtois, F. Heymans, and D. L. Parnas. “Concurrent Control with "Readers" and "Writers"”. In: *Commun. ACM* 14.10 (1971), pp. 667–668. DOI: 10.1145/362759.362813.

[32] S. Cranen, B. Luttik, and T. A. C. Willemse. “Evidence for Fixpoint Logic”. In: *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*. Ed. by S. Kreutzer. Vol. 41. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 78–93. DOI: 10.4230/LIPIcs.CSL.2015.78.

[33] S. Cranen, B. Luttik, and T. A. C. Willemse. “Proof Graphs for Parameterised Boolean Equation Systems”. In: *CONCUR*. Ed. by P. R. D’Argenio and H. C. Melgratti. Vol. 8052. LNCS. Springer, 2013, pp. 470–484. DOI: 10.1007/978-3-642-40184-8_33.

[34] P. Crouzen and F. Lang. “Smart Reduction”. In: *FASE*. Ed. by D. Giannakopoulou and F. Orejas. Vol. 6603. LNCS. Springer, 2011, pp. 111–126. DOI: 10.1007/978-3-642-19811-3_9.

[35] Á. Darvas, R. Hähnle, and D. Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. Ed. by D. Hutter and M. Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer, 2005, pp. 193–209. DOI: 10.1007/978-3-540-32004-3_20.

[36] D. Dechev, P. Pirkelbauer, and B. Stroustrup. “Lock-Free Dynamically Resizable Arrays”. In: *OPODIS 2006*. Ed. by A. A. Shvartsman. Vol. 4305. LNCS. Springer, 2006, pp. 142–156. DOI: 10.1007/11945529_11.

[37] L. P. Deutsch. “An interactive program verifier”. In: (1973).

[38] D. Dice and A. Kogan. “BRAVO - Biased Locking for Reader-Writer Locks”. In: *2019 USENIX*. Ed. by D. Malkhi and D. Tsafrir. USENIX Association, 2019, pp. 315–328.

[39] T. van Dijk. “Oink: An Implementation and Evaluation of Modern Parity Game Solvers”. In: *TACAS*. Ed. by D. Beyer and M. Huisman. Vol. 10805. LNCS. Springer, 2018, pp. 291–308. DOI: 10.1007/978-3-319-89960-2_16.

[40] T. van Dijk and J. van de Pol. “Multi-core symbolic bisimulation minimisation”. In: *Int. J. Softw. Tools Technol. Transf.* 20.2 (2018), pp. 157–177. DOI: 10.1007/s10009-017-0468-z.

[41] T. van Dijk and J. van de Pol. “Sylvan: multi-core framework for decision diagrams”. In: *Int. J. Softw. Tools Technol. Transf.* 19.6 (2017), pp. 675–696. DOI: 10.1007/s10009-016-0433-2.

[42] L. Doyen and J. Raskin. “Antichain Algorithms for Finite Automata”. In: *TACAS*. Ed. by J. Esparza and R. Majumdar. Vol. 6015. LNCS. Springer, 2010, pp. 2–22. DOI: 10.1007/978-3-642-12002-2_2.

[43] Á. T. Eiríksson and K. L. McMillan. “Using Formal Verification/Analysis Methods on the Critical Path in System Design: A Case Study”. In: *CAV*. Ed. by P. Wolper. Vol. 939. LNCS. Springer, 1995, pp. 367–380. DOI: 10.1007/3-540-60045-0_63.

[44] S. Enevoldsen, K. G. Larsen, and J. Srba. “Abstract Dependency Graphs and Their Application to Model Checking”. In: *TACAS 2019*. Ed. by T. Vojnar and L. Zhang. Vol. 11427. LNCS. Springer, 2019, pp. 316–333. DOI: 10.1007/978-3-030-17462-0_18.

[45] R. Erkens and M. Laveaux. “Adaptive Non-linear Pattern Matching Automata”. In: *Log. Methods Comput. Sci.* 17.4 (2021). DOI: 10.46298/lmcs-17(4:21)2021.

[46] W. Fokkink, J. Pang, and J. Pol. “Cones and foci: A mechanical framework for protocol verification”. In: *Formal Methods Syst. Des.* 29.1 (2006), pp. 1–31. DOI: 10.1007/s10703-006-0004-3.

[47] H. Gao, J. Groote, and W. Hesselink. “Lock-free dynamic hash tables with open addressing”. In: *Distributed Comput.* 18.1 (2005), pp. 21–42. DOI: 10.1007/s00446-004-0115-2.

[48] H. Gao, J. Groote, and W. Hesselink. “Lock-free parallel and concurrent garbage collection by mark&sweep”. In: *Sci. Comput. Program.* 64.3 (2007), pp. 341–374. DOI: 10.1016/j.scico.2006.10.001.

[49] Z. Gao, C. Bird, and E. T. Barr. "To type or not to type: quantifying detectable bugs in JavaScript". In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by S. Uchitel, A. Orso, and M. P. Robillard. IEEE / ACM, 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75.

[50] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". In: *STTT* 15.2 (2013), pp. 89–107. DOI: 10.1007/s10009-012-0244-z.

[51] H. Garavel, F. Lang, and L. Mounier. "Compositional Verification in Action". In: *FMICS*. Ed. by F. Howar and J. Barnat. Vol. 11119. LNCS. Springer, 2018, pp. 189–210. DOI: 10.1007/978-3-030-00244-2_13.

[52] T. Gibson-Robinson, G. H. Broadfoot, G. Carvalho, P. J. Hopcroft, G. Lowe, S. Nogueira, C. O'Halloran, and A. Sampaio. "FDR: From Theory to Industrial Application". In: *Concurrency, Security, and Puzzles*. Vol. 10160. LNCS. Springer, 2017, pp. 65–87.

[53] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. "FDR3 - A Modern Refinement Checker for CSP". In: *TACAS*. Ed. by E. Ábrahám and K. Havelund. Vol. 8413. LNCS. Springer, 2014, pp. 187–201. DOI: 10.1007/978-3-642-54862-8_13.

[54] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. "FDR3: a parallel refinement checker for CSP". In: *Int. J. Softw. Tools Technol. Transf.* 18.2 (2016), pp. 149–167. DOI: 10.1007/s10009-015-0377-y.

[55] R. J. van Glabbeek. Personal Communication, 7 January 2019. 2019.

[56] R. J. van Glabbeek, B. Luttik, and N. Trčka. "Branching Bisimilarity with Explicit Divergence". In: *Fundam. Inform.* 93.4 (2009), pp. 371–392. DOI: 10.3233/FI-2009-109.

[57] R. J. van Glabbeek and W. P. Weijland. "Branching Time and Abstraction in Bisimulation Semantics". In: *J. ACM* 43.3 (1996), pp. 555–600. DOI: 10.1145/233551.233556.

[58] R. Glabbeek, S. Luttik, and N. Trcka. "Branching Bisimilarity with Explicit Divergence". In: *Fundam. Informaticae* 93.4 (2009), pp. 371–392. DOI: 10.3233/FI-2009-109.

[59] R. J. van Glabbeek. "A Branching Time Model of CSP". In: *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*. Ed. by T. Gibson-Robinson, P. J. Hopcroft, and R. Lazić. Vol. 10160. LNCS. Springer, 2017, pp. 272–293. DOI: 10.1007/978-3-319-51046-0_14.

[60] R. J. van Glabbeek. “The Linear Time - Branching Time Spectrum II”. In: *CONCUR*. Ed. by E. Best. Vol. 715. LNCS. Springer, 1993, pp. 66–81. DOI: 10.1007/3-540-57208-2_6.

[61] A. O. Gomes and A. Butterfield. “Modelling the Haemodialysis Machine with Circus”. In: *ABZ 2016*. Ed. by M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró. Vol. 9675. LNCS. Springer, 2016, pp. 409–424. DOI: 10.1007/978-3-319-33600-8_34.

[62] S. Graf, B. Steffen, and G. Lüttgen. “Compositional Minimisation of Finite State Systems Using Interface Specifications”. In: *Formal Asp. Comput.* 8.5 (1996), pp. 607–616. DOI: 10.1007/BF01211911.

[63] J. F. Groote and M. R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. ISBN: 9780262027717.

[64] J. Groote and J. Keiren. “Tutorial: Designing Distributed Software in mCRL2”. In: *FORTE*. Ed. by K. Peters and T. Willemse. Vol. 12719. LNCS. Springer, 2021, pp. 226–243. DOI: 10.1007/978-3-030-78089-0_15.

[65] J. Groote and J. Springintveld. “Focus points and convergent process operators: a proof strategy for protocol verification”. In: *J. Log. Algebraic Methods Program.* 49.1-2 (2001), pp. 31–60. DOI: 10.1016/S1567-8326(01)00010-8.

[66] J. F. Groote, K. H. J. Jilissen, M. Laveaux, P. H. M. van Spaendonck, and T. A. C. Willemse. “Using the Parallel ATerm Library for Parallel Model Checking and State Space Generation”. In: *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Ed. by N. Jansen, M. Stoelinga, and P. van den Bos. Vol. 13560. LNCS. Springer, 2022, pp. 306–320. DOI: 10.1007/978-3-031-15629-8_16.

[67] J. F. Groote and B. Lisser. “Computer assisted manipulation of algebraic process specifications”. In: *ACM SIGPLAN Notices* 37.12 (2002), pp. 98–107. DOI: 10.1145/636517.636531.

[68] J. F. Groote and F. Moller. “Verification of Parallel Systems via Decomposition”. In: *CONCUR*. Ed. by R. Cleaveland. Vol. 630. LNCS. Springer, 1992, pp. 62–76. DOI: 10.1007/BFb0084783.

[69] J. F. Groote and T. A. C. Willemse. “Model-checking processes with data”. In: *Sci. Comput. Program.* 56.3 (2005), pp. 251–273.

[70] J. F. Groote and T. A. C. Willemse. “Parameterised Boolean equation systems”. In: *Theor. Comput. Sci.* 343.3 (2005), pp. 332–369. DOI: 10.1016/j.tcs.2005.06.016.

[71] *Haskell Language*. https://www.haskell.org/.

[72] M. Hennessy and H. Lin. "Symbolic Bisimulations". In: *Theor. Comput. Sci.* 138.2 (1995), pp. 353–389. DOI: 10.1016/0304-3975(94)00172-F.

[73] J. Herkert, J. Borenstein, and K. W. Miller. "The Boeing 737 MAX: Lessons for Engineering Ethics". In: *Sci. Eng. Ethics* 26.6 (2020), pp. 2957–2974. DOI: 10.1007/s11948-020-00252-y.

[74] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008. ISBN: 978-0-12-370591-4.

[75] M. Herlihy. "The art of multiprocessor programming". In: *PODC 2006*. Ed. by E. Ruppert and D. Malkhi. ACM, 2006, pp. 1–2. DOI: 10.1145/1146381.1146382.

[76] W. Hesselink and J. Groote. "Wait-free concurrent memory management by Create and Read until Deletion (CaRuD)". In: *Distributed Comput.* 14.1 (2001), pp. 31–39. DOI: 10.1007/PL00008924.

[77] W. H. Hesselink. "Invariants for the Construction of a Handshake Register". In: *Inf. Process. Lett.* 68.4 (1998), pp. 173–177. DOI: 10.1016/S0020-0190(98)00158-6.

[78] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[79] T. Hoare. *Null References: The Billion Dollar Mistake*. 2009.

[80] W. C. Hsieh and W. E. Weihl. "Scalable Reader-Writer Locks for Parallel Systems". In: *Proceedings of the 6th International Parallel Processing Symposium*. Ed. by V. K. Prasanna and L. H. Canter. IEEE Computer Society, 1992, pp. 656–659. DOI: 10.1109/IPPS.1992.222989.

[81] M. Huth, J. H. Kuo, and N. Piterman. "Fatal Attractors in Parity Games". In: *FOSSACS*. Ed. by F. Pfenning. Vol. 7794. LNCS. Springer, 2013, pp. 34–49. DOI: 10.1007/978-3-642-37075-5_3.

[82] R. Huybers and A. Laarman. "A Parallel Relation-Based Algorithm for Symbolic Bisimulation Minimization". In: *VMCAI 2019*. Ed. by C. Enea and R. Piskac. Vol. 11388. LNCS. Springer, 2019, pp. 535–554. DOI: 10.1007/978-3-030-11245-5_25.

[83] D. N. Jansen, J. F. Groote, J. J. A. Keiren, and A. Wijs. "An O(m log n) algorithm for branching bisimilarity on labelled transition systems". In: *TACAS*. Ed. by A. Biere and D. Parker. Vol. 12079. LNCS. Springer, 2020, pp. 3–20. DOI: 10.1007/978-3-030-45237-7_1.

[84] H. A. de Jong and P. A. Olivier. “Generation of abstract programming interfaces from syntax definitions”. In: *J. Log. Algebraic Methods Program.* 59.1-2 (2004), pp. 35–61. DOI: 10.1016/j.jlap.2003.12.002.

[85] S. T. Q. Jongmans, D. Clarke, and J. Proença. “A procedure for splitting data-aware processes and its application to coordination”. In: *Sci. Comput. Program.* 115-116 (2016), pp. 47–78. DOI: 10.1016/j.scico.2014.02.017.

[86] M. Jurdziński and R. Lazić. “Succinct progress measures for solving parity games”. In: *LICS*. IEEE Computer Society, 2017, pp. 1–9. DOI: 10.1109/LICS.2017.8005092.

[87] P. C. Kanellakis and S. A. Smolka. “CCS Expressions, Finite State Processes, and Three Problems of Equivalence”. In: *Inf. Comput.* 86.1 (1990), pp. 43–68. DOI: 10.1016/0890-5401(90)90025-D.

[88] G. Kant and J. van de Pol. “Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games”. In: *GRAPHITE*. Vol. 99. EPTCS. 2012, pp. 50–65.

[89] G. Kant and J. van de Pol. “Generating and Solving Symbolic Parity Games”. In: *GRAPHITE*. Ed. by D. Bosnacki, S. Edelkamp, A. Lluch-Lafuente, and A. Wijs. Vol. 159. EPTCS. 2014, pp. 2–14. DOI: 10.4204/EPTCS.159.2.

[90] A. Lankamp. https://github.com/cwi-swat/aterms/blob/master/shared-objects/src/shared/SharedObjectFactory.java. 2009.

[91] M. Laveaux. *Downloadable sources and benchmarks for the experimental validation*. https://doi.org/10.5281/zenodo.3449420. 2019.

[92] M. Laveaux. *Downloadable sources for the case study*. 2021. DOI: 10.5281/zenodo.5091850.

[93] M. Laveaux, J. F. Groote, and T. A. C. Willemse. “Correct and Efficient Antichain Algorithms for Refinement Checking”. In: *Log. Methods Comput. Sci.* 17.1 (2021).

[94] M. Laveaux, W. Wesselink, and T. A. C. Willemse. “On-The-Fly Solving for Symbolic Parity Games”. In: *TACAS 2022*. Ed. by D. Fisman and G. Rosu. Vol. 13244. LNCS. Springer, 2022, pp. 137–155. DOI: 10.1007/978-3-030-99527-0_8.

[95] M. Laveaux and T. A. C. Willemse. “Decomposing Monolithic Processes in a Process Algebra with Multi-actions”. In: *ICE 2021*. Ed. by J. Lange, A. Mavridou, L. Safina, and A. Scalas. Vol. 347. EPTCS. 2021, pp. 57–76. DOI: 10.4204/EPTCS.347.4.

[96] K. R. M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *LPAR-16*. Ed. by E. M. Clarke and A. Voronkov. Vol. 6355. LNCS. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[97] R. Liu, H. Zhang, and H. Chen. “Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks”. In: *2014 USENIX*. Ed. by G. Gibson and N. Zeldovich. USENIX Association, 2014, pp. 219–230.

[98] S. Luttik. “Description and formal specification of the Link Layer of P1394”. In: *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design*. Ed. by I. Lovrek. University of Zagreb, Croatia, 1997, pp. 43–56.

[99] R. Mateescu and M. Sighireanu. “Efficient on-the-fly model-checking for regular alternation-free mu-calculus”. In: *Sci. Comput. Program.* 46.3 (2003), pp. 255–281. DOI: 10.1016/S0167-6423(02)00094-1.

[100] R. McNaughton. “Infinite Games Played on Finite Graphs”. In: *Ann. Pure Appl. Logic* 65.2 (1993), pp. 149–184. DOI: 10.1016/0168-0072(93)90036-D.

[101] Microsoft. *Microsoft Exchange Server Remote Code Execution Vulnerability CVE-2021-26855*. 2021.

[102] D. M. Miller. “Multiple-Valued Logic Design Tools”. In: *ISMVL*. IEEE Computer Society, 1993, pp. 2–11. DOI: 10.1109/ISMVL.1993.289589.

[103] R. Milner. *A Calculus of Communicating Systems*. Vol. 92. LNCS. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3.

[104] R. Milner. “Calculi for Synchrony and Asynchrony”. In: *Theor. Comput. Sci.* 25 (1983), pp. 267–310. DOI: 10.1016/0304-3975(83)90114-7.

[105] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[106] J. P. Nielsen and S. Karlsson. “A scalable lock-free hash table with open addressing”. In: *PPoPP 2016*. Ed. by R. Asenjo and T. Harris. ACM, 2016, 33:1–33:2. DOI: 10.1145/2851141.2851196.

[107] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[108] J. O’Leary. “Formal verification in Intel CPU design”. In: *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE Computer Society, 2004, p. 152. DOI: 10.1109/MEMCOD.2004.1459841.

[109] S. Owre, J. M. Rushby, and N. Shankar. “PVS: A Prototype Verification System”. In: *CADE-11s*. Ed. by D. Kapur. Vol. 607. LNCS. Springer, 1992, pp. 748–752. DOI: 10.1007/3-540-55602-8_217.

[110] R. Paige and R. E. Tarjan. “Three Partition Refinement Algorithms”. In: *SIAM J. Comput.* 16.6 (1987), pp. 973–989. DOI: 10.1137/0216062.

[111] J. Pang, W. J. Fokkink, R. F. H. Hofman, and R. Veldema. “Model checking a cache coherence protocol of a Java DSM implementation”. In: *J. Log. Algebraic Methods Program.* 71.1 (2007), pp. 1–43. DOI: 10.1016/j.jlap.2006.08.007.

[112] R. Paval. “Modeling and Verifying Concurrent Data Structures”. MA thesis. Eindhoven University of Technology, 2018.

[113] A. Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[114] J. van de Pol and M. Timmer. “State Space Reduction of Linear Processes Using Control Flow Reconstruction”. In: *ATVA*. Ed. by Z. Liu and A. P. Ravn. Vol. 5799. LNCS. Springer, 2009, pp. 54–68. DOI: 10.1007/978-3-642-04761-9_5.

[115] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. “Concurrent tries with efficient non-blocking snapshots”. In: *ACM SIGPLAN*. Ed. by J. Ramanujam and P. Sadayappan. ACM, 2012, pp. 151–160. DOI: 10.1145/2145816.2145836.

[116] F. Ranzato and F. Tapparo. “Generalizing the Paige-Tarjan algorithm by abstract interpretation”. In: *Inf. Comput.* 206.5 (2008), pp. 620–651. DOI: 10.1016/j.ic.2008.01.001.

[117] M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. ISBN: 978-3-642-32026-2. DOI: 10.1007/978-3-642-32027-9.

[118] D. Remenska, T. A. C. Willemse, K. Verstoep, J. Templon, and H. E. Bal. “Using model checking to analyze the system behavior of the LHC production grid”. In: *Future Gener. Comput. Syst.* 29.8 (2013), pp. 2239–2251. DOI: 10.1016/j.future.2013.06.004.

[119] A. Rensink and W. Vogler. “Fair testing”. In: *Inf. Comput.* 205.2 (2007), pp. 125–198. DOI: 10.1016/j.ic.2006.06.002.

[120] J. Romijn and J. Springintveld. “Exploiting Symmetry in Protocol Testing”. In: *FORTE*. Ed. by S. Budkowski, A. R. Cavalli, and E. Najm. Vol. 135. IFIP Conference Proceedings. Kluwer, 1998, pp. 337–352. DOI: 10.1007/978-0-387-35394-4_29.

[121] A. W. Roscoe. “Model-checking CSP”. In: *A Classical Mind: essays in Honour of C. A. R. Hoare*. Ed. by A. W. Roscoe. Prentice Hall International (UK) Ltd., 1994. Chap. 21, pp. 353–378.

[122] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010. ISBN: 978-1-84882-257-3. DOI: 10.1007/978-1-84882-258-0.

[123] *Rust Programming Language*. https://www.rust-lang.org/.

[124] L. Sanchez, W. Wesselink, and T. A. C. Willemse. “A Comparison of BDD-Based Parity Game Solvers”. In: *GandALF*. Ed. by A. Orlandini and M. Zimmermann. Vol. 277. EPTCS. 2018, pp. 103–117. DOI: 10.4204/EPTCS.277.8.

[125] C. Shann, T. Huang, and C. Chen. “A Practical Nonblocking Queue Algorithm Using Compare-and-Swap”. In: *ICPADS*. IEEE Computer Society, 2000, pp. 470–475. DOI: 10.1109/ICPADS.2000.857731.

[126] A. D. Stasio, A. Murano, and M. Y. Vardi. “Solving Parity Games: Explicit vs Symbolic”. In: *CIAA*. Ed. by C. Câmpeanu. Vol. 10977. LNCS. Springer, 2018, pp. 159–172. DOI: 10.1007/978-3-319-94812-6_14.

[127] A. Sterling. *88% Of People Will Abandon An App Because Of Bugs*. 2017.

[128] C. Stirling and D. Walker. “Local Model Checking in the Modal mu-Calculus”. In: *Theor. Comput. Sci.* 89.1 (1991), pp. 161–177. DOI: 10.1016/0304-3975(90)90110-4.

[129] Y. Sun and G. E. Blelloch. “Implementing parallel and concurrent tree structures”. In: *ACM SIGPLAN*. Ed. by J. K. Hollingsworth and I. Keidar. ACM, 2019, pp. 447–450. DOI: 10.1145/3293883.3302576.

[130] K. Tai and P. V. Koppol. “An Incremental Approach to Reachability Analysis of Distributed Programs”. In: *IWSSD*. Ed. by J. C. Wileden. IEEE Computer Society, 1993, pp. 141–151. DOI: 10.1109/IWSSD.1993.315504.

[131] K. Tai and P. V. Koppol. “Hierarchy-based incremental analysis of communication protocols”. In: *ICNP*. IEEE Computer Society, 1993, pp. 318–325. DOI: 10.1109/ICNP.1993.340896.

[132] A. S. Tanenbaum and D. Wetherall. *Computer networks, 5th Edition*. Pearson, 2011. ISBN: 0132553171.

[133] T. C. D. Team. *The Coq Proof Assistant*. 2022. DOI: 10.5281/zenodo.5846982.

[134] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.

[135] I. Umar, O. J. Anshus, and P. H. Ha. "GreenBST: Energy-Efficient Concurrent Search Tree". In: *Parallel Processing*. Ed. by P. Dutot and D. Trystram. Vol. 9833. LNCS. Springer, 2016, pp. 502–517. DOI: 10.1007/978-3-319-43659-3_37.

[136] S. van der Vegt and A. Laarman. "A Parallel Compact Hash Table". In: *MEMICS 2011*. Ed. by Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, and D. Antos. Vol. 7119. LNCS. Springer, 2011, pp. 191–204. DOI: 10.1007/978-3-642-25929-6_18.

[137] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. "More Anti-chain Based Refinement Checking". In: *ICFEM*. Ed. by T. Aoki and K. Taguchi. Vol. 7635. LNCS. Springer, 2012, pp. 364–380. DOI: 10.1007/978-3-642-34281-3_26.

[138] W. Wesselink and T. A. C. Willemse. "Evidence Extraction from Parameterised Boolean Equation Systems". In: *ARQNL 2018*. Ed. by C. Benzmüller and J. Otten. Vol. 2095. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 86–100.

[139] M. D. Wulf, L. Doyen, T. A. Henzinger, and J. Raskin. "Antichains: A New Algorithm for Checking Universality of Finite Automata". In: *CAV*. Ed. by T. Ball and R. B. Jones. Vol. 4144. LNCS. Springer, 2006, pp. 17–30. DOI: 10.1007/11817963_5.

[140] W. Zielonka. "Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees". In: *Theor. Comput. Sci.* 200.1-2 (1998), pp. 135–183. DOI: 10.1016/S0304-3975(98)00009-7.

Summary

Accelerated Verification of Concurrent Systems

Computers are ubiquitous in our daily lives, ranging from the small chip in your debit card to the data centres that form the foundation of the internet. The behaviour of these computers is described by software. Implementing this software correctly, which means that the actual behaviour is equal to the intended behaviour, is notoriously difficult. This is especially true for concurrent systems where interaction between computers must also be taken into account.

A key ingredient to ensure the correctness of software is to specify its behaviour in a way that can be analysed formally. In our setting we specify this behaviour as processes using a process algebra. Model checking is a technique that considers a model of the specification and automatically verifies that the specified behaviour matches the intended behaviour, which must also be specified. The behavioural model of a specification is in our case defined by a state space consisting of all possible states of the system and a transition relation between states that defines the (nondeterministic) choices in each state. A major obstacle for the verification of practical systems is the size of these state spaces. In this dissertation we investigate several ways to tackle this problem.

We identify several issues pertaining to the soundness and performance in existing antichain-based refinement algorithms and propose new, correct, antichain-based algorithms. Refinement is a model checking technique that checks whether the behaviour of one process refines the behaviour of another process. It is the basis of a stepwise development methodology in which the correctness of a system can be established by proving, or computing, that a system refines its specification.

We define a decomposition technique to effectively obtain the state space of a so called monolithic process. A monolithic process is a single recursive equation with data parameters, which only uses non-determinism, action prefixing, and recursion. For the decomposition we can show that a composition of these processes is strongly bisimilar

to the monolithic process under a suitable synchronisation context. Minimising the resulting processes before determining their composition can be used to derive a state space that is smaller than the one obtained by a monolithic exploration.

Another model checking technique considers a specification and a property written in a formal language and decides whether the specification satisfies this property, which is called the model checking problem. Parity games are a typical encoding used for this model checking problem. We formalise how to use on-the-fly solving techniques of parity games can be applied during the exploration process, and show that this can help to decide the decision problem more efficiently by terminating early.

Finally, we propose a thread-safe term library that can be used as the basis for concurrent model checking algorithms, including the previously described ones. Terms are one of the fundamental data structures used for computing and therefore are also extensively used in the toolset where our techniques have been implemented. To this end we define a new efficient multiple-reader/single write mutual exclusion algorithm that has been shown to be correct used model checking. Using the new library in an existing state space generation tool, very substantial speed ups can be obtained.

Samenvatting

Versnelde Verificatie van Gelijktijdige Systemen

Computers zijn onmisbaar is ons dagelijks leven, van de kleine microprocessoren in een pinpas tot de datacentra die de fundering van het internet vormen. Het gedrag van deze computers wordt daarbij door programmatuur beschreven. Deze programmatuur correct implementeren, waarbij correct betekent dat het daadwerkelijke gedrag dezelfde is als het bedoelde gedrag, is welbekend een lastig probleem. Dit geldt zeker voor systemen die met elkaar communiceren, omdat daarbij ook rekening moet worden gehouden met het gedrag dat door de communicatie wordt beïnvloed.

Een essentiële methode om de correctheid van programmatuur te garanderen is door het gedrag op te schrijven op een manier dat het formeel bestudeerd kan worden. Dit gedrag wordt in ons geval beschreven als een proces in een zogeheten proces algebra. Model verificatie is een techniek die een specificatie model bekijkt en automatisch verifieert of het beschreven gedrag identiek is aan het vereiste gedrag, waarbij dat laatste ook moet worden beschreven. Het specificatie gedrag wordt in dit geval definieert als een toestandsruimte bestaande uit alle mogelijke toestanden van het system en een transitie relatie tussen toestanden die de niet-deterministische keuze in elke toestand aanduidt. Een groot obstakel voor de verificatie van praktische systemen is de grootte van deze toestandsruimtes. In dit proefschrift onderzoeken we verschillende manieren om dit specifieke probleem aan te pakken.

We identificeren meerdere problemen met betrekking tot de correctheid en efficiëntie in bestaande op zogeheten ‘antichain’ gebaseerde verfijning algoritmes in de literatuur en beschrijven nieuwe, correcte, ‘antichain’ gebaseerde verfijning algoritmes. Verfijning is een model verificatie techniek die beslist of het gedrag van een systeem het gedrag van een ander model in zekere zin verfijnt. Deze techniek is de basis voor een stapsgewijze programmatuur ontwikkel techniek waarbij de correctheid van het systeem wordt gegeven door een verfijning van componenten en hun bijbehorende specificatie te bewijzen, of te berekenen.

We definiëren een decompositie techniek om op een effectieve manier de toestandsruimte van een zogenaamd monolithisch proces te verkrijgen. Een monolithisch proces is een proces dat bestaat uit een enkele recursieve vergelijking met data parameters, dat alleen niet deterministische keuze en actie prefix gebruikt. Voor deze decompositie kunnen we laten zien dat een specifieke compositie van de componenten equivalent is aan de oorspronkelijke monolithisch proces onder sterke bisimulatie voor een bepaalde synchronisatie context. Het reduceren van de resulterende processen voordat de compositie wordt berekend kan ertoe leiden dat de verkregen toestandsruimte direct kleiner is dan de toestandsruimte die verkregen wordt uit de monolithische exploratie.

Een andere model verificatie techniek beschouwt een specificatie van het systeem en een formule geschreven in een wiskundige taal en probeert te verifiëren of de formule geldt voor deze specificatie, dit is het "model checking" probleem. Pariteit spellen worden typisch gebruikt om dit probleem te coderen. We formaliseren hoe een oplos methode gebruikt kan worden tijdens het exploratie proces, en laten zien dat op deze manier de exploratie mogelijk eerder gestopt kan worden omdat de oplossing van het spel eerder gevonden kan worden.

Ten slot formaliseren wij een term programmatuur bibliotheek die door meerdere threads ('draden') tegelijk benaderd kan worden, en die gebruikt kan worden als basis voor parallelle model verificatie algoritmes, inclusief de algoritmes die eerder zijn beschreven. Termen zijn een fundamentele data structuur voor de berekeningen en wordt daarom uitsluitend gebruikt in de applicatie waarin we onze technieken hebben geïmplementeerd. Voor efficiëntie definiëren we een nieuwe protocol om meerdere lezers en een enkele schrijver exclusieve toegang te verlenen dat correct bewezen is met behulp van model verificatie technieken voor eindige instanties. Deze nieuwe term programmatuur bibliotheek is gebruikt om substantiële versnellingen te verkrijgen in het toestandsruimte exploratie proces.

Curriculum Vitae

Maurice Laveaux was born on the 9th of January 1994 in Geleen, The Netherlands. After finishing vwo in 2012 at Graaf Huyn College in Geleen, The Netherlands, he studied Computer Science and Engineering at Eindhoven University of technique in Eindhoven, The Netherlands. In 2018 he graduated cum laude within the Formal System Analysis group on solving real-valued parameterised Boolean equation systems. From 2018 he started a PhD project at Eindhoven University of Technology at Eindhoven, The Netherlands, of which the results are presented in this dissertation.

Titles in the IPA Dissertation Series since 2019

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Cosimulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10