Improving Scalability of Exploratory Model Checking

Alexandre Boulgakov

Thesis submitted to the University of Oxford
for the degree of Doctor of Philosophy
in
Computer Science

Department of Computer Science
and Merton College,
University of Oxford

Michaelmas Term 2016

Supervised by Prof. A. W. Roscoe

Declaration

The contents of this thesis are all my own work, except where otherwise stated. The views and opinions expressed herein are mine and not necessarily those of any other person or body unless so attributed.

This thesis builds on top of the existing foundation of FDR, and relies on both the code base of FDR3 [GRABR15] and on the insights present in this and previous versions (cited in Section 2.2) of FDR.

The Haskell-based compiler presented in Section 3.1.1 was partly implemented by Thomas Gibson-Robinson. Specifically, he implemented compilation of the non-process expressions and the author implemented compilation of processes. The LLVM-based lazy compiler presented in Chapter 3 was partly integrated into FDR3 by Thomas Gibson-Robinson; this includes an extension to the parser as well as to the execution pipeline that allows it to pass to the compiler unmodified. The lazy compiler itself as highlighted in Figure 3.2 was implemented entirely by the author, but used the existing lazy enumeration functionality as a base for the custom lazy enumerator necessary for the lazy compiler.

The Naïve Iterative Refinement method for strong bisimulation was previously implemented in FDR3. The rest of the implementations of bisimulation in FDR3 presented in Chapter 4 were implemented by the author. Integration of these algorithms into FDR3 was assisted by Thomas Gibson-Robinson and Philip Armstrong. Some of the work presented in this thesis has led to joint publications. Specifically, much of the

work on bisimulation has been presented in [BGRR14, BGRR16].

Citation: Alexandre Boulgakov (2017), Improving Scalability of Exploratory Model Checking, D.Phil. Thesis, University of Oxford, Department of Computer Science, Oxford, UK.

Keywords: CSP, FDR, Model-checking, Verification, Labelled transition systems, Lazy compilation, Bisimulation, Strong bisimulation, Delay bisimulation, Weak bisimulation

This thesis has been typeset using LaTeX 2$_\varepsilon$ and references were compiled using BibTeX.

Copyright ©, by Alexandre Boulgakov, 2017.
All rights reserved.

No part of the material protected by this copyright notice may be reproduced or utilised in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the copyright owner.

Printed in the United Kingdom.

Abstract

Improving Scalability of Exploratory Model Checking

Alexandre Boulgakov
Merton College, Oxford

Michaelmas Term 2016

As software and hardware systems grow more complex and we begin to rely more on their correctness and reliability, it becomes exceedingly important to formally verify certain properties of these systems. If done naïvely, verifying a system can easily require exponentially more work than running it, in order to account for all possible executions. However, there are often symmetries or other properties of a system that can be exploited to reduce the amount of necessary work. In this thesis, we present a number of approaches that do this in the context of the CSP model checker FDR. CSP is named for *Communicating Sequential Processes*, or parallel combinations of state machines with synchronised communications. In the FDR model, the component processes are typically converted to explicit state machines while their parallel combination is evaluated lazily during model checking. Our contributions are motivated by this model but applicable to other models as well.

We first address the scalability of the component machines by proposing a *lazy compiler* for a subset of CSP_M selected to model parameterised state machines. This is a typical case where the state space explosion can make model checking impractical, since the size of the state space is exponential in the number and size of the parameters. A lazy approach to evaluating these systems allows only the reachable subset of the state space to be explored. As an example, in studying security protocols, it is common to model an intruder parameterised by knowledge of each of a list of facts; even a relatively small 100 facts results in an intractable 2^{100} states, but the rest of the system can ensure that only a small number of these states are reachable.

Next, we address the scalability of the overall combination by presenting novel algorithms for bisimulation reduction with respect to *strong bisimulation*, *divergence-respecting delay bisimulation*, and *divergence-respecting weak bisimulation*. Since a parallel composition is related to the Cartesian product of its components, performing a relatively time-consuming bisimulation reduction on the components can reduce its size significantly; an efficient bisimulation algorithm is therefore very desirable.

This thesis is motivated by practical implementations, and we discuss an implementation of each of the proposed algorithms in FDR. We thoroughly evaluate their performance and demonstrate their effectiveness.

Acknowledgements

I cannot overstate my gratitude to my supervisor, Bill Roscoe, who has been immensely helpful both in his technical and non-technical roles. On the technical side, his expertise resulted in much useful collaboration in the form of ideas, discussion, and more proof-reading than anyone else could put up with. Besides that, he got me more involved with the scientific community and helped with funding for my final months. I am particularly grateful to him for our regular meetings and his insistence that I continue working even when I felt stuck.

I would like to thank Michael Goldsmith, Ben Worrell, and Steve Schneider for their feedback at various stages of my thesis. Michael and Ben's early feedback, which helped set the direction of the thesis towards making a more valuable contribution to the field, was particularly useful, and Michael and Steve's feedback during my defence kept it from slipping off that trajectory.

I am very grateful to Thomas Gibson-Robinson for patiently explaining the inner workings of FDR and for our many cake-fueled discussions, and to Philippa Hopcroft for joining these discussions.

I am immeasurably grateful to my parents, who instilled in me a love for academia and computer science from a very young age, who encouraged me to pursue this DPhil, and who offered much needed advice, ranging from the best graph algorithms to use to how many pairs of socks I should bring to a conference.

I would also like to thank my friends for distracting and supporting me when I

needed it, and for keeping my mind fresh with stimulating discussions on a wider range of topics than I might have otherwise considered. I owe a special thanks to Dong Woo for her support when I first moved to this new country. I am also very grateful to the various dance societies that have touched me in one way or another, or more precisely, to the people I met there: particularly the Oxford University Dancesport Club and Team, the Oxford Lindy Hoppers, and Swing Dance Luxembourg, whom I met only briefly but who recognised me as a fellow dancer straight away and made me feel very welcome at my first conference.

A big thanks also goes to Rosie for keeping me sane during the final push. Without her support, encouragement, and understanding, this thesis might not have been completed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the adage goes, "to err is human, but to really foul things up requires a computer," and this is particularly true of the complex software and hardware systems designed by large teams that are so common today. An error can arise from any step of the process: the original design might be flawed, or the implementation of one of its components, or the bridges that connect the components. Errors can even arise in short pieces of code that seem trivially correct at first glance.

A notable example of this is the Zune 30 leap year bug [BBC08] that caused thousands of Zune media players to hang and heat up for 24 hours on the 366th day of 2008. The code that caused the problem is listed in Figure 1.1. Normally, it subtracts 365 or 366 days, depending on whether the year is a leap year, until there are 365 or fewer days left. It might appear at first glance to be correct, but a closer examination reveals that when days is 366 on a leap year, it does not subtract anything (per the innermost `if`) but does not exit the outermost loop. Fortunately, this bug only affected the devices in question and resolved itself the next day.

When the bug is in a well-known protocol, however, all systems implementing the protocol are affected, potentially resulting in much higher costs. For example, the Needham-Schroeder Public-Key Protocol [NS78], which provides mutual authentication over an insecure channel, proposed in 1978, was thought to be secure until a vulnerability to a man-in-the-middle attack was discovered nearly 20 years later [Low96].

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

Figure 1.1: The code responsible for the Zune leap year bug.

It is unknown how many systems that used the protocol were actually compromised after the discovery, but it serves to show that even a protocol tested with years of industry experience might be flawed.

It is clear that some bugs, especially those dealing with corner cases can stand up to a moderate amount of scrutiny, creating a need for automated verification of at least those systems where the potential costs of failure are enough to justify the time and resources spent on verification. It often takes a lot more computational power to prove the correctness of a system than to run it once, because it is only correct if every possible execution is correct. As a result, software systems pushing the limits of currently available computational power cannot be verified directly. However, a model of the system that abstracts away some of the details may well be verifiable. Model checking is the field that is concerned with checking certain properties of a model, which may model a complete system faithfully or introduce abstractions as desirable.

Meaning of correctness. Desirable properties vary from system to system. A program that prints "Hello, world!" is a valid *hello world* program, but not a valid calculator, for example. So, in addition to the model that is being examined, a spec-

ification must be provided. There are several popular approaches to expressing the specification.

Some tools provide hard-coded general-purpose specifications that the user cannot alter. For example, PEVerify [MG01] proves that its input meets certain type safety requirements and Spin [Hol04] provides a built-in deadlock freedom check and unreachable code detection. Other tools extract a specification from the model. For example, a C# compiler generates an overflow check for integer arithmetic outside an unchecked context (Section 14.5.13 of [ECM06]). Spin extracts and verifies assertions embedded in code and the .NET Code Contracts [Fäh10] library supports the design-by-contract paradigm to extract the pre- and post-conditions of a method. These approaches are not very versatile since the former only allows a small and fixed set of properties to be verified (a program that prints "Hello, world!" over and over again passes the deadlock freedom check but is still not a valid calculator) and the latter can allow arbitrary properties (depending on the implementation) of a given state to be expressed, but provides no mechanism for restricting the path a program might take to reach that state.

Some tools (such as Spin and Spot [DLLF+16]) provide a way to express path properties as formulas in one of a number of temporal logics, such as μ-calculus [Pra81], LTL [Pnu77], CTL [CE81], or CTL* [EH86]. Such formulas can be effective in expressing properties of the system, but they require the user to learn a new language for specifications, and this language can be difficult to read, increasing the chance of bugs in the specification.

Other tools allow the specification to be written in the same language as the model, with the semantics of correctness being those of *refinement*, where an implementation is correct with respect to the specification if it exhibits a subset of the behaviours allowed by the specification. This has a range of benefits in addition to only requiring knowledge of one language, including refinement-based development, where the implementation is derived from the specification, and stepwise refinement checking, where development is done in stages and each stage is proven to implement the previous stage possibly with

additional requirements. This is the approach taken by FDR, and will be the primary focus of this thesis.[1]

Model languages. As with specifications, there is a lot of variety in the languages used for the implementation. These can be executable languages, such as Common Intermediate Language [ECM01] or C, or abstract languages such as CSP (used by FDR), PROMELA (used by Spin), or Dafny [Lei10]. There is obvious demand for verifying systems written in executable languages, since this would allow production systems to be verified; in fact, even some tools that primarily work with abstract languages, such as Spin, support inline C code to simplify verification of embedded systems.

On the other hand, a dedicated modelling language has the benefit of a strong and platform-independent mathematical foundation that is not often present in languages designed with other goals. Additionally, such a language can introduce constructs that cannot be efficiently implemented in production but that can be useful for abstraction or specification. For example, many modelling languages include a *nondeterministic choice* operator that can be useful for *don't-care* specifications or for modelling the potential inputs to a component while abstracting away the system that provides them.

Mechanisms of model checking. The approaches to model checking can be classified as *implicit-state* or *explicit-state* based on how they represent visited states. In the former, the states are represented by symbolic structures such as formulas or binary decision diagrams. An example of such an approach is bounded model checking, which reduces a model checking problem to an instance of SAT and solves it efficiently with a SAT solver. By contrast, in explicit-state model checking, the state space of a process is represented by a labelled transition system or a similarly explicit structure and explored directly.

Both approaches have their strengths. An explicit transition system can be explored

[1] Although FDR also provides shortcuts for common operations such as deadlock, livelock, and determinism checking, these are implemented internally by refinement checking against a static specification or one derived from the input process.

in time linear in its size, while implicit approaches such as those relying on the NP-complete SAT can be slower in the size of their input. However it is important to note that their inputs are not directly comparable as the size of an implicit representation depends not only on the size of the state space, but also on its structure. For example, a system with n-bit states all of which are reachable can be represented compactly by the formula $\Phi(b_1, ...b_n) = true$ even for large n, but if a random selection of those states are not reachable, the formula can become considerably larger.

Work on SymFDR [POR09], an extension of FDR using SAT-solving, has shown that bounded refinement checking can improve performance significantly for some combinatorially complex examples that violate their specification, but in general FDR outperformed SymFDR. This indicates that while there is still a niche for implicit-state approaches it is also important to work on improving the scalability of explicit-state approaches, which will therefore be the subject of this thesis.

State explosion problem. A significant obstacle to explicit-state model checking is that the entire state space must be recorded. Symbolic model checking can suffer from growth of the state space as well, but it is more significant in explicit-state model checking where memory for state storage usually becomes the limiting factor in large checks. Even where the memory usage is not prohibitive, it has an indirect effect on speed due to typical computer architectures employing a memory hierarchy ranging from small amounts of fast memory (such as processor registers and caches), to increasingly larger but slower memories (ranging from main memory to flash storage to hard disks to network storage).

It is not difficult to write a model that grows out of control. Systems are typically built of components connected in parallel with some sort of communication between them, and combinatorics allows the sizes of the component processes to get multiplied. This can result in state spaces whose size is exponential in the length of the input program, and it is this exponential growth that is known as the *state explosion problem.*

In rich enough modelling languages, it is also possible to write large component

processes by means of parameterisation. For example, we can define a class of counter processes $P(n)$ that each output n and become $P(n+1)$ for all n (in CSP, this can be expressed in just one line, $P(n) = n \rightarrow P(n+1)$); each of these processes has an infinite number of states. Finite but large parameterised processes are also possible. A common example is modelling a cryptographic intruder (see, e.g., [RG97, RSG+01] for existing approaches and Section 3.7.1 for our novel approach) that can know any of a number of facts. The natural way to write an intruder process is to parameterise it by the facts it knows (either as a set parameter or as a number of Boolean parameters) with transitions from every state for each fact it can learn, each leading to a state where it has learned that fact as well as anything it can now infer. However, this results in a state space exponential in the number of facts, though many of these states might be unvisited in the final composition: if the protocol being analysed prevents the intruder from learning just one of the facts, the state space will be halved, and other constraints will reduce the state space further.

Although there are many approaches to combatting this problem, there are still cases where a human can do better, and these cases vary from tool to tool, depending on the state space reduction techniques the tool uses. While components executing in parallel are traditionally viewed as interleaved (in the absence of synchronisation), that is, with a total order imposed on their actions, partial order methods instead leave actions whose relative order is irrelevant unordered [God95]. This can provide an exponential reduction in the number of states where there are a lot of asynchronously parallel processes. Partial order reduction techniques differ in the amount of information they track (corresponding to different properties that can be preserved).

If the system is composed of several components, each component can be expressed as a labelled transition system (LTS) and then compressed to another LTS using an arbitrary semantics-preserving function before composition. Though the work done on each component is added, the reductions in state space are multiplied. Partial order reduction is one such compression, but it is typically applied to the final composition rather than the individual components since there are more effective techniques that

can be applied to small components. For example, a component can be reduced to the smallest component strongly bisimilar to it in most process algebras (the ones where strong bisimulation equivalence holds) or to the smallest component bisimilar to the input under other semantics-preserving notions of bisimulation ([Par81, Mil81, BGRR16]) such as divergence-respecting delay bisimulation or divergence-respecting weak bisimulation. In process algebras with an invisible action (τ), all subgraphs that are strongly connected under can be reduced to a single divergent state.

There are also approaches that do not reduce the number of states but reduce their impact. When verifying a system that is thought to have a counterexample, different exploration orders will visit different numbers of states before finding the counterexample. They do not improve the worst-case performance (since all states might still have to be visited), but choosing the right order can help for systems that are known to have a counterexample. For example, the user or the tool might select between breadth-first search, depth-first search, and bounded depth-first search if the length of the counterexample is known. Intelligent probing is an automated approach that aims to visit the states most likely to generate a counterexample first.

Different methods of storing the state space can have different performance characteristics, with a trade-off between time and space. Spin uses a deterministic finite automaton to represent the visited state space [HP99]. FDR uses a B-tree with LZ4, LZ4HC, or Zip compression to reduce the storage requirements to 0.2-0.3 of the uncompressed B-tree.

Finally, when none of these automatic solutions are sufficient, the user is expected to know the tool and design the model around it. For example, FDR uses a breadth-first search for finding new states (in order to find the shortest counterexample first) but a depth-first search for livelock checking. So an FDR user working on a problem with lots of symmetry and a long safety counterexample might recode the problem from a safety specification to a liveness specification.

1.1 Outline

In Chapter 2 we give a detailed overview of the CSP process algebra, the CSP refinement checker FDR, and related tools, along with references for further reading on these topics. In Chapter 3 we present a compiler for CSP_M, a machine-readable variant of CSP, with lazy evaluation semantics. We cover the motivation for such a compiler, overview the LLVM framework we used to implement a prototype, provide the details of the major components, and present and analyse a number of case studies. We then suggest several directions for future work. In Chapter 4 we discuss a number of algorithms for bisimulation reduction. We give an overview of the state of the art before presenting novel algorithms for computing the maximal divergence-respecting delay and weak bisimulation of an LTS. Lastly, we present a thorough analysis of the performance of current algorithms as well as those presented in this thesis. Finally, in Chapter 5 we summarise the main results of the thesis, although the reader is advised to look at Section 3.9 and Section 4.5 for more details.

Chapter 2

Background

In this chapter, we will give a background of the CSP process algebra (Section 2.1) and the CSP refinement checker FDR (Section 2.2), as well as an overview of related tools in Section 2.3.

2.1 CSP

The Communicating Sequential Processes (CSP) process algebra developed by Hoare [Hoa78, Hoa85] provides the theoretical underpinnings of the machine-readable CSP_M language used throughout this thesis. This section gives a basic introduction to CSP (Section 2.1.1), its operational (Section 2.1.3) and denotational semantics (Section 2.1.4), and CSP_M (Section 2.1.2). A more detailed description can be found in [Ros98] or [Ros10].

CSP has constructs for creating sequential processes that can perform a sequence of *events*, as well as for building larger systems through the use of parallel combinations, possibly synchronising the involved processes on a set of events. A process can communicate a member of the set of syntactic events Σ or one of the special events τ and $\checkmark$. The invisible event τ represents internal progress. It cannot be observed and does not interact with the environment. The event $\checkmark$ represents *termination* and will not be discussed here in depth.

2.1.1 Syntax

STOP. The built-in *STOP* process can perform no events.

Prefix. The process $a \rightarrow P$ offers the environment the event $a \in \Sigma$ and then behaves like P. This is the main means of generating events, since most of the other operators do not generate events on their own, but rather control or modify the events of their component processes.

We sometimes structure events by sending them along a *channel*. For example, $c.3$ denotes the value 3 being sent along the channel c; channels can also allow multiple dot-separated values such as $d.1.true$. When using channels with the prefix operator, a ! can be used in place of the dot to indicate output along the channel, and a ? can be used to indicate input. Thus, $c!3 \rightarrow P$ is equivalent to $c.3 \rightarrow P$. Input such as $c?x \rightarrow Q(x)$ is accomplished by offering the environment a choice (external choice as described below) of all events of the form $c.x$ and binding x accordingly; thus, $c?x \rightarrow Q(x)$ where c is a channel of integers is equivalent to $c.0 \rightarrow Q(0) \,\Box\, c.1 \rightarrow Q(1) \,\Box\, ...$

External choice. The process $P \,\Box\, Q$ offers the environment the choice of the events offered by P and by Q. It is *resolved* by the first visible (i.e., non-τ) action performed by either process, and behaves like that process from that point.

Internal choice. $P \sqcap Q$ also behaves as either P or Q, but in this case the choice is not offered to the environment but performed non-deterministically.

Interleave. $P \,|||\, Q$ is the simplest parallel operator, running P and Q in parallel but enforcing no synchronisation between them.

Generalised parallel. $P \underset{A}{\parallel} Q$ allows P and Q to run in parallel, forcing synchronisation on events in A and arbitrarily interleaving events not in A. Other parallel operators can be expressed using generalised parallel. For example. $P \,|||\, Q$ is equivalent to $P \underset{\emptyset}{\parallel} Q$.

Hiding and Renaming. These two operators do not affect the structure of a process, but change the events it performs. $P \setminus A$ behaves as P but hides any events from A by transforming them into the internal event τ. $P[\![R]\!]$, behaves as P but renames the events according to the relation R. Hence, if P can perform a, then $P[\![R]\!]$ can perform each b such that $(a, b) \in R$, where the choice (if more than one such b) is left to the environment (like $\Box$).

Other operators. CSP has a wide range of operators that allow processes to be expressed naturally and succinctly. Moreover, as demonstrated in [Ros11, Ros15, Ros10], there is a large class of *CSP-like* operators that are consistent with the theory of CSP and can be defined using the existing CSP operators.

Two examples of CSP's more advanced operators are *interrupt* and *exception*. $P \triangle Q$ initially behaves like P but allows Q to *interrupt* at any point and perform a visible event, at which point P is discarded and the process behaves like Q. $P \ \Theta_A \ Q$ initially behaves like P, but if P ever performs an event from A, P is discarded and $P \ \Theta_A \ Q$ behaves like Q.

CSP also supports process *termination* signalled by the special event $\checkmark$. This is introduced by the special process *Skip*, which immediately terminates and is equivalent to $\checkmark \rightarrow STOP$. *Sequential composition*, denoted by $P; \ Q$, runs P until it terminates at which point Q is run. More precisely, $P; \ Q$ initially behaves as P, but if P performs a $\checkmark$, $P; \ Q$ performs a τ to the state Q.

2.1.2 CSP$_M$

The previous section covers the blackboard versions of CSP operators, and intentionally omits details of the mathematical notation that can be used to define sets, relations, or named and parameterised processes. This is because there are no formal rules regarding these and the usual mathematical notation is used.

However, while this works well for humans, the machine-readable CSP$_M$ is necessary to support tools such as FDR. This adds ASCII versions of all operators and a

Haskell-like functional language to formalise aspects of the language such as parameterised processes or mathematical expressions. Details of this can be found in the FDR manual[1]. We will use blackboard CSP where appropriate throughout this thesis and introduce CSP_M constructs as needed.

2.1.3 Operational Semantics

As an explicit model checker, FDR uses the operational semantics of CSP. This section will cover the notation for CSP operational semantics, as well as present the rules for several operators. A more comprehensive presentation can be found in [Ros98, Ros10].

The operational semantics of a CSP process are modelled as a *labelled transition system* (LTS) where the nodes are *process states* and transitions are labelled by events from $\Sigma \cup \{\tau\}$. In the context of the formal semantics, a process state is an abstract identifier. However, intuitively, it corresponds to a process, modelled by the same LTS but with it as the start node.

Definition 2.1.1. A *labelled transition system* (LTS) is a tuple $(N, n_0, \Sigma, \longrightarrow)$ where N is a set of nodes, $n_0 \in N$ is the initial node, Σ is a set of events, τ is a designated *invisible* event not in Σ, $\Sigma^\tau = \Sigma \cup \{\tau\}$, and $\longrightarrow \subseteq N \times \Sigma^\tau \times N$ is a labelled transition relation (with $p \xrightarrow{a} q$ indicating a transition from p to q with action a). The following shorthand is used:

- $\mathit{initials}(m) = \{e \mid \exists n \cdot m \xrightarrow{e} n\}$ denotes m's initial events;
- $\mathit{afters}(m) = \{(e, n) \mid m \xrightarrow{e} n\}$ denotes m's directly enabled transitions;
- $m\Uparrow \Leftrightarrow \exists m_0, m_1, \ldots \cdot m_0 = m \wedge \forall\, i \cdot m_i \xrightarrow{\tau} m_{i+1}$ denotes *divergence*, i.e. an infinite series of internal τ actions corresponding to *livelock*.

The usual way of defining the operational semantics of a CSP process P is to define Σ and n_0 (corresponding to the process P) explicitly, to define $\longrightarrow$ using *Structured Operational Semantics* (SOS) style rules, and to implicitly define N as the set of nodes

[1] `http://www.cs.ox.ac.uk/projects/fdr/manual/cspm.html`

reachable from n_0 through the transitive closure of $\longrightarrow$. As an example, the operational semantics of the exception operator are defined by:

$$\frac{P \xrightarrow{a} P'}{P \ \Theta_A \ Q \xrightarrow{a} Q} \ a \in A$$

$$\frac{P \xrightarrow{b} P'}{P \ \Theta_A \ Q \xrightarrow{b} P' \ \Theta_A \ Q} \ b \notin A$$

$$\frac{P \xrightarrow{\tau} P'}{P \ \Theta_A \ Q \xrightarrow{\tau} P' \ \Theta_A \ Q}$$

The interesting rule is the first, which specifies that if P performs an event $a \in A$, then $P \ \Theta_A \ Q$ can perform the event a and behave like Q. The other two rules allow P to perform any other event including τ.

An argument P of a CSP operator Op is **on** iff there is a rule of Op that allows P to perform an action immediately. P is **off** iff no such rule exists. For example, the left argument of the exception operator is **on** since the rules specify that it can perform visible events, whilst the right argument is **off** since there are no rules that allow it to perform events.

The SOS style of operational semantics is far more expressive than is required to give an operational semantics to CSP, and indeed can define operators which, for a variety of reasons, make no sense in CSP models. As pointed out in [Ros10], it is possible to re-formulate CSP's semantics in the highly restricted *combinator* style of operational semantics, which largely concentrates on the relationships between events of argument processes and those of the constructed system. This style says, *inter alia*, that only **on** arguments can influence events, that any τ action of an **on** argument must be allowed to proceed freely, and that an argument process has changed state in the result state if and only if it has participated in the action. Cloning of **on** arguments is not permitted. Any language with a combinator operational semantics can be translated to CSP with a high degree of faithfulness [Ros10] and is compositional over every CSP model. FDR3 is designed so that it can readily be extended to such *CSP-like* languages.

Definition 2.1.2. The combinator operational semantics for a particular operator is

a set of triples of the form (ϕ, x, R) where

- ϕ is a partial function from the indices of the **on** arguments to Σ. It represents the actions that each of the **on** arguments performs to contribute to an overall action, and corresponds to the input line of an SOS-style rule. If there are $M > 1$ **on** arguments, we will write ϕ as an M-tuple whose components are members of Σ or $\cdot$ to indicate a component does not participate in this action. If there is one **on** argument, we will write a member of Σ or $\cdot$, and we will use – if there are no **on** arguments.

- $x \in \Sigma^{\tau\checkmark}$ is the overall action that the operator performs. This corresponds to the event of the transition in the output line of an SOS-style rule.

- R defines the *format* of the result state and corresponds to the right-hand side of the output line of an SOS-style rule. This is a piece of CSP syntax with argument indices as placeholders. An index **i** stands for the corresponding process after it has performed the event indicated by ϕ. Namely, if $i \in dom(\phi)$, then the process P_i has performed an action $P_i \xrightarrow{\phi(i)} P_i'$, and **i** represents P_i'. Otherwise, **i** represents P_i. For convenience, we will omit this term if it is isomorphic to the input format.

All indices are 1-based.

We will now present the combinator-style rules for each of the operators introduced in Section 2.1.1.

STOP. STOP has no **on** arguments and no rules.

Prefix. $a \rightarrow P$ has a single rule: (–, a, **1**).

External choice. $P \,\Box\, Q$ has the rules $((a, \cdot),a,\mathbf{1})$ and $((\cdot, a),a,\mathbf{2})$ for each $a \in \Sigma$.

Internal choice. $P \sqcap Q$ has the rules (–,τ,**1**) and (–,τ,**2**).

Interleave. $P \,|||\, Q$ has the rules $((a, \cdot),a)$ and $((\cdot, a),a)$ for each $a \in \Sigma$.

Generalised parallel. $P \underset{A}{\parallel} Q$ has the synchronising rule $((a, a), a)$ for each $a \in A$, and the non-synchronising rules $((a, \cdot), a)$ and $((\cdot, a), a)$ for each $a \notin A$.

Hiding. $P \setminus A$ has the hiding rule (a,τ) for each $a \in A$ and the non-hiding rule (a,a) for each $a \notin A$.

Renaming. $P[\![R]\!]$ has the rule (a,b) for each $(a, b) \in R$.

Interrupt. $P \triangle Q$ has the interrupting rule $((\cdot, a),a,\mathbf{2})$ for each $a \in \Sigma$ and the non-interrupting rule $((a, \cdot),a)$ for each $a \in \Sigma$.

Exception. $P \ \Theta_A \ Q$ has the exception rule $(a,a,\mathbf{2})$ for each $a \in A$ and the non-exception rule (a,a) for each $a \notin A$.

2.1.4 Denotational Semantics

While the operational semantics captures the behaviour of a process completely, a more high-level view can be useful for reasoning about processes. The denotational semantics of CSP offer precisely this. There are a number of commonly used semantic models which offer varying levels of abstraction.

Of these, the simplest is the *finite traces* model $\mathcal{T}$. A *trace* is a sequence of visible events that the process can communicate, and $\mathcal{T}$ represents a process by its set of finite traces. Denotationally, these can be calculated using a set of rules including $\mathit{traces}(STOP) = \{\langle\rangle\}$, $\mathit{traces}(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathit{traces}(P)\}$, and $\mathit{traces}(P \ \Box \ Q) = \mathit{traces}(P) \cup \mathit{traces}(Q)$. Since a trace is a sequence of non-τ transition labels on a finite walk through the LTS corresponding to the process, traces can also be computed from the operational semantics of a process.

However, there are some properties of a process that $\mathcal{T}$ does not capture. For example, it cannot observe *livelock* (or *divergence*), in which a process performs an infinite sequence of hidden τ events that are not recorded as part of the trace, or potential *deadlock* since a process might have both a deadlocking and a non-deadlocking

behaviour with the same initial subsequence (the processes $STOP \sqcap a \rightarrow STOP$ and $a \rightarrow STOP$ have the same traces, but the former can deadlock immediately, without first performing any visible events). As shown in [Ros09], $\mathcal{T}$ is the *coarsest* non-trivial model for CSP, that is, any congruence for CSP that distinguishes at least two processes can also distinguish any processes distinguished by $\mathcal{T}$. There is an infinite hierarchy of finer models, which can capture deadlock, livelock, or other properties.

In particular, the most commonly used model for CSP is the *failures-divergences* model $\mathcal{N}$, representing a process as a set of *failures* and a set of *divergences*. A failure couples each trace with a set of events that the process can permanently refuse, and allows deadlock to be detected as a failure of the form (s, Σ) for some trace s. A divergence is a trace after which the process can perform an unbounded number of τs. In addition, this model assumes a process can perform any behaviour after a potential divergence, and accordingly augments the divergences set for a process P with $\{s \hat{} t \mid s \in \mathit{divergences}(P) \wedge t \in \Sigma^*\}$ and the failures set with $\{(s, X) \mid s \in \mathit{divergences}(P) \wedge X \subseteq \Sigma\}$.

2.1.4.1 Refinement

Each denotational model allows specifications of a process to be expressed using existing logical formalisms, of which there is a wide range. Such logics do not necessarily capture all observable properties (which are all properties we might like to use in a specification) of CSP or allow unobservable properties to be expressed (branching time logics such as CTL, for example, are not consistent with CSP's notion of linear observations of a single execution).

An alternative way to express specifications is with CSP itself, by creating a process with all behaviours satisfying the specification that are possible for a process to have, and testing for *refinement*. A process *Impl refines Spec* in a given model M, written $\mathit{Spec} \sqsubseteq_M \mathit{Impl}$, exactly when it has a subset of the possible behaviours of *Spec*. For example, for the traces model $\mathcal{T}$, $\mathit{Spec} \sqsubseteq_T \mathit{Impl}$ is equivalent to $\mathit{traces}(\mathit{Impl}) \subseteq \mathit{traces}(\mathit{Spec})$. The use of such *characteristic* processes for specification

ensures that the set of expressible specifications is exactly those that can be observed, and frees the user from learning an additional language to use for specification.

2.2 FDR

FDR [Ros94, RGG+95, GRABR15] is a commercial[2] model checker for CSP$_M$. Most importantly, it supports refinement checking. Additionally, it provides a number of built-in checks such as determinism or deadlock freedom; internally these are implemented as refinement checks. This section introduces FDR with a particular focus on the technical details of FDR3, the version used throughout this thesis. Figure 2.1 illustrates the overall structure of FDR3, and each of the components will be covered in turn.

As in most tools that work with scripts, the input CSP$_M$ file first goes through a parser that converts it to an in-memory representation. This is then type-checked to ensure that everything is consistently typed at the level of the functional language and annotates each construct with its type. The parser and type-checker are part of the open-source Haskell library `libcspm` [Uni13].

2.2.1 Evaluator

The next step is the *evaluator*, which is also part of `libcspm`. This is the component responsible for removing the functional language and converting the CSP$_M$ to a tree of CSP operator applications. For example, if P was the CSP$_M$ expression:

```
if true then c?x -> STOP else STOP
```

this would evaluate to:

$$c.0 \longrightarrow STOP \;\Box\; c.1 \longrightarrow STOP$$

[2]FDR was originally released by Formal Systems (Europe) Ltd. The most recent version is available from `https://www.cs.ox.ac.uk/projects/fdr`.

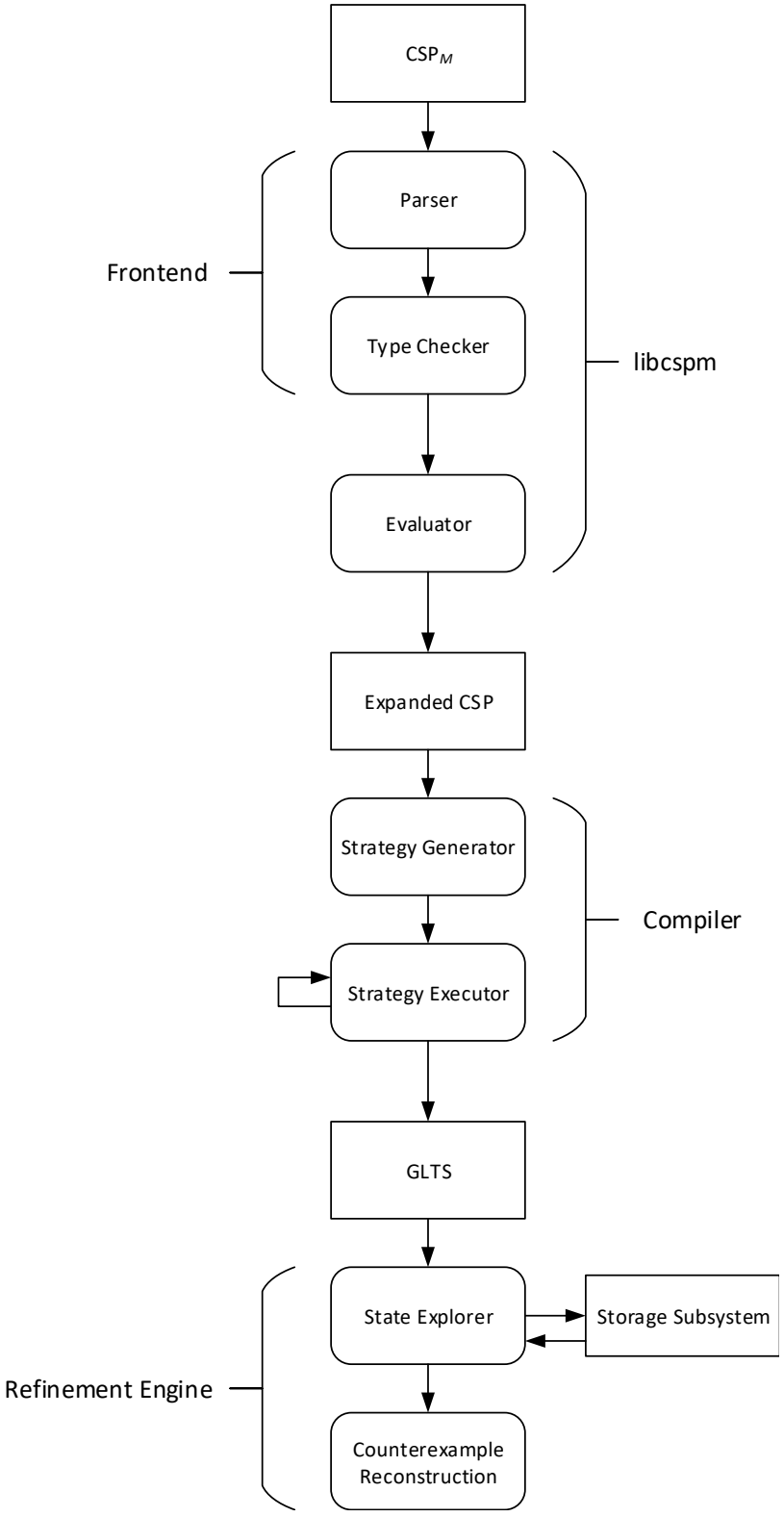


Figure 2.1: Overall structure of FDR3.

Notice that the functional language has been removed: all that remains is a tree of trivial operator applications, as follows.

Definition 2.2.1. A *syntactic process* P is generated according to the grammar:

$$P ::= Operator(P_1, \ldots, P_M) \mid N$$

where the P_i are also syntactic processes, *Operator* is any CSP operator (e.g. external choice, prefix, etc.) and N is a *process name.* A *syntactic process environment* Γ is a function from process name to syntactic process such that $\Gamma(N)$ is never a process name.[3].

For example, the recursive process

$$P = c.0 \longrightarrow STOP \;\Box\; c.1 \longrightarrow P$$

is represented as:

$$\Gamma(P) = ExtChoice(Prefix_{c.0}(STOP), Prefix_{c.1}(P))$$

where $STOP$ and P are process names.

It is important to note that in current versions of FDR, a CSP_M expression must be evaluated fully before being passed on to the compiler, which only works with syntactic processes. In the common case where the resulting machine is much larger than its syntactic representation and most of the time is spent in the state explorer, this works well and avoids the overhead of frequent two-way communication between the evaluator and compiler. However, this eager evaluation can in some cases perform more work than is necessary, particularly with large component processes that will only have a fraction of their states and transitions visited over the course of a refinement check:

[3]This avoids the possibility of *immediate recursions* such as $P := Q$. These are permitted in CSP_M, but the evaluator essentially removes them.

consider at the extreme the infinite-state process $P(0)$ where:

$$P(i) = a \rightarrow P(i+1)$$

placed in the highly restricted environment:

$$P(0) \underset{\{a\}}{\parallel} STOP$$

While infinite-state processes are best avoided for other reasons, it can sometimes be desirable to create processes with a finite, but intractable number of states placed in parallel with other processes that ensure only a reasonably sized fraction of their state space is visited during the refinement check. In Chapter 3, we design a lazy compiler for a subset of CSP_M that can efficiently deal with such processes.

2.2.2 Compiler

The expanded CSP produced by the evaluator is fed into the *compiler*, which converts it to an LTS which is used to represent CSP processes during refinement checks. In order to support various features (most importantly, the *state-compressions* such as *normalisation*), FDR internally represents processes as *generalised labelled transition systems* (GLTSs), rather than LTSs. These differ from LTSs in that the individual states can be labelled with properties according to the semantic model in use. For example, if the failures model is being used, a GLTS would allow states to be labelled with refusals.

Definition 2.2.2. A *generalised labelled transition system* is a tuple (G, Λ, λ) where G is an LTS with node set N, Λ is a set of node labels, and $\lambda : N \rightarrow \Lambda$ is a total function labelling each node.

Due to the generality of CSP in which operators can be combined in almost arbitrary ways, the primary challenge for the compiler is to decide which of FDR3's internal representations of GLTSs (which have various trade-offs) should be used to

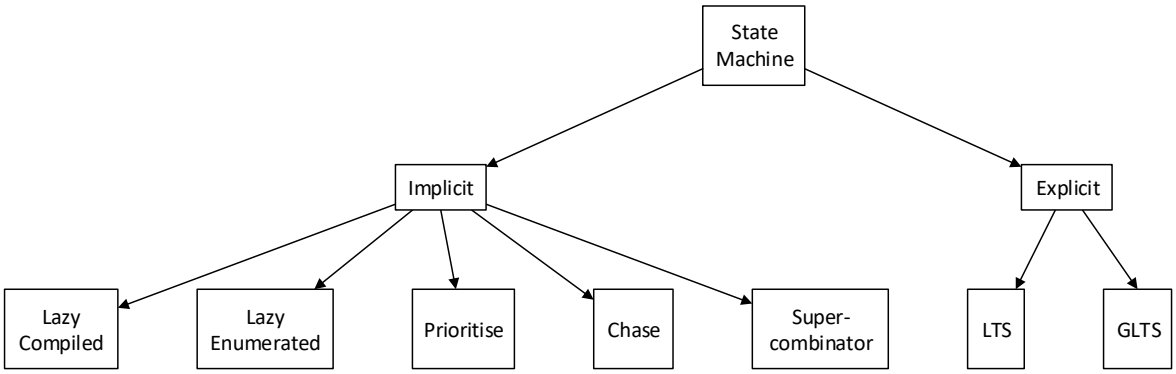


Figure 2.2: Machine types used by FDR3.

represent each syntactic process. In order to allow the processes to be represented efficiently, FDR3 has a number of different GLTS types, and a number of different ways of constructing each GLTS.

As shown in Figure 2.2, FDR3 can represent GLTSs *explicitly* or *implicitly*. The former representation stores the states and transitions of a machine explicitly, and uses a corresponding amount of memory. An explicit GLTS is simply a standard graph data structure. States in an explicit GLTS are process states whilst the transitions are stored in a sorted list.

In contrast, implicit machines compute their transitions on the fly and only require the states to be stored during a refinement check. Equally, it takes longer to calculate the transitions of an implicit machine than the corresponding explicit machine. For models that require minimal acceptance sets, these can also be computed on the fly.

The most common type of implicit machine is a *supercombinator* machine, which represents the GLTS by a series of component GLTSs along with a list of rules to combine the transitions of the components. Rules can also be split into *formats*, which are sets of rules, and can also specify that a component machine should be *restarted* in its initial state. These rules describe how to combine the actions of P and Q into actions of the whole machine and correspond to the combinator operational semantics described in Definition 2.1.2.

The component GLTSs of a supercombinator machine can themselves be supercombinator machines, which allows complex trees of operator applications to be represented

directly using their combinator operational semantics. Alternatively, to avoid the cost of navigating this tree each time a transition needs to be computed, FDR supports an optimisation where an entire tree of combinator applications is *supercompiled* into a *supercombinator*, which is a combinator that does not necessarily correspond to a single CSP operator application.

For example, a supercombinator for $P \mid\mid\mid Q$ is the combinator given in Section 2.1.3 and consists of the components $\langle P, Q \rangle$ and the rules:

$$\begin{aligned} &\{(\langle 1 \mapsto a \rangle, a) \mid a \in \alpha P \cup \{\tau\}\} \\ &\cup \{(\langle 2 \mapsto a \rangle, a) \mid a \in \alpha Q \cup \{\tau\}\} \end{aligned}$$

where αX is the alphabet of the process X (i.e. the set of events it can perform). Details of FDR3's representation are given in [GRABR15]. The supercombinator for $(P \mid\mid\mid Q) \mid\mid\mid R$, on the other hand, combines the rules for $P \mid\mid\mid Q$ and $\cdot \mid\mid\mid R$ to produce:

$$\begin{aligned} &\{(\langle 1 \mapsto a \rangle, a) \mid a \in \alpha P \cup \{\tau\}\} \\ &\cup \{(\langle 2 \mapsto a \rangle, a) \mid a \in \alpha Q \cup \{\tau\}\} \\ &\cup \{(\langle 3 \mapsto a \rangle, a) \mid a \in \alpha R \cup \{\tau\}\}. \end{aligned}$$

The *prioritise* and *chase* machines are specialised for their respective operators and will not be discussed here. A *lazy enumerated* machine is used internally to wrap another machine but convert its states to integers. The *lazy compiled* machine is a new type introduced to support lazy compilation as presented in Chapter 3.

There are several different *strategies* that FDR3 can use to construct explicit or supercombinator machines from syntactic processes, differing in the type of processes that they can support (e.g. some cannot support recursive processes), the time they take to execute, and the type of the resulting GLTS. *Low level* is the simplest strategy and supports any process. An explicit LTS is constructed simply by directly applying CSP's operational semantics. *High level* compiles a process to a supercombinator. This is not

able to compile *recursive* processes, such as $P := a \rightarrow P$. The supercombinator rules are directly constructed using the operational semantics of CSP. *Mixed level* is a hybrid of the low- and high-level strategies where, intuitively, non-recursive parts of processes are compiled as per the high-level strategy whilst recursive parts are compiled as per the low-level strategy. *Recursive high level* compiles to a supercombinator machine and allows some recursive processes to be compiled at the high level.

The *strategy generator* is responsible for determining how to compile a given process. The input to the compilation algorithm is a syntactic process environment (Definition 2.2.1) and the output is a list of strategies that specify how each syntactic processes should be compiled. The algorithm guarantees to produce a strategy such that executing the strategy yields a valid GLTS that corresponds to the input process. The algorithm also uses heuristics to attempt to reduce the time and memory usage during the subsequent refinement check. In particular, the present strategy generator aims to compile component processes to explicit machines with the expectation that their transitions will be visited enough times during exploration of the larger system to merit caching them, and that the components are sufficiently small that this does not create memory pressure. The specific strategies used are described in more detail in [GRABR15]. The *strategy executor* then applies the computed strategy to the input syntactic process and outputs a GLTS.

Compressions. Since version 2, FDR supports a variety of *compressions*[RGG+95] which can be used to cut the state space of a system or subsystem. These can be requested in a CSP_M script explicitly, or applied automatically in the strategy generator. An important class of compressions is *bisimulations*. A number of bisimulation compressions and their efficient implementation in FDR3 are discussed in detail in Chapter 4.

2.2.3 State Explorer

Once the specification and implementation processes have been converted into GLTSs, the *state explorer* can finally perform the refinement check. As described in [GRABR15], refinement checking proceeds by performing a search over the implementation, checking that every reachable state is compatible with some state of the specification for which there is a trace on which both can be reached[4]. A parallel breadth-first search is performed since this produces a minimal counterexample when the check fails. The breadth-first search is organised as an iterative algorithm, where each iteration checks nodes on a single *ply* (also known as *level*). For example, the first ply will contain just the root node, whilst the second ply contains all states that are reachable from the root node in a single step, etc. Sets of state pairs are stored in B-Trees (rather than, e.g., hash tables) in order to allow checks to remain efficient when exceeding the size of the available RAM.

Due to this explicit-state approach, the limiting factor is typically the space used, which is closely related to the size of the state space. More precisely, in the case of a supercombinator machine, it is the size of the *reachable* state space that is relevant. In general, the best bound on the size of the parallel composition of an M-state process with an N-state process is $M \times N$ states, though communication between the processes can restrict the actual state space to a subset of this. Nevertheless, parallel compositions make it quite easy to create extremely large processes. This can often be mitigated by spending some time compressing component processes; while the benefits for each such process might not be large, they can have a considerable impact on the composition.

2.2.4 Counterexample Reconstruction

If the state explorer finds a state pair such that the implementation violates the specification, FDR can present a *counterexample* to the user illustrating the specific be-

[4] A process called *normalisation* ensures that there is exactly one state of the specification for any given trace.

haviours of the implementation and the specification leading up to this state pair. *Counterexample reconstruction* is this final step that allows the behaviours to be computed (often relying on a separate, much quicker refinement check) and split into the behaviours of component processes that enabled them.

2.3 Related Work

Spin. Spin [Hol04] is a system targeting the verification of multi-threaded software modelled in PROMELA. Verification is typically performed with respect to specifications expressed in linear temporal logic (LTL), as Büchi automata, or as omega-regular properties; additionally, verifiable invariants can be embedded in the model itself. The model and specifications are compiled into a C-based executable verifier, which allows arbitrary C code to be embedded in the model.

By default, Spin stores states in a hash table, and is thus limited by main memory. However, it can alternatively store states using a minimised automaton [HP99] or use *bitstate hashing* [Hol98] to compute a highly accurate probabilistic approximation using a minimal amount of space. Additionally, Spin supports cluster verification [HJG08], allowing both more memory and more computing resources to be assigned to a check. Spin uses *on-the-fly* verification in both the local and cluster modes, allowing the state space to be explored lazily.

CADP. The CADP (Construction and Analysis of Distributed Processes) Toolbox [GLMS13] is a collection of tools supporting input in a range of process algebras and a number of manipulations and checks on the resulting systems. It supports equivalence checking modulo bisimulation relations as well as model checking with respect to various logics. A number of explicit and implicit verification algorithms are included. Many of the algorithms depend on main memory, which can be a limitation, but it has been tested on systems with hundreds of millions of states.

Some of the input formats include rich, structured process algebras with support for parallel composition of subsystems. However, these are typically expanded to explicit

LTSs before further processing, in contrast to FDR's two-level approach to compilation detailed in Section 2.2.

The included `BCG_MIN` tool can minimise an LTS with respect to one of a number of bisimulations. We include it in a detailed comparison of bisimulation approaches in Section 4.4.

mCRL2. The mCRL2 toolset [CGK+13] provides both an eponymous language based on μCRL and a collection of tools for reasoning about systems described in that language. It supports model checking with respect to the regular modal μ-calculus using an implicit approach based on *parameterised Boolean equation systems*. Additionally, it allows exporting transition systems for use with other tools, such as the CADP toolbox. A typical mCRL2 model of a distributed system uses parallel composition, but similarly to CADP, mCRL2 requires that models be *linearised* to a series of condition-action-effect rules before further processing.

Chapter 3

Lazy Compiler

As discussed in Section 2.2.1, FDR can be excessively strict when evaluating component processes. In this chapter, we present and evaluate a lazy compiler that can be used in some cases where eager evaluation is not desirable. This compiler targets LLVM, a platform designed as a compiler back-end, and is fully integrated into FDR. We have limited the language it accepts to a subset of CSP_M that omits certain features we found to be problematic. However, the lazy compiler is intended to supplement rather than replace the existing evaluator, and the restrictions only apply to code the user actively chooses to compile lazily.

We discuss the motivation for the lazy compiler and our selected approach in more detail in Section 3.1, including a brief overview of our attempt to support the full CSP_M language and use Haskell as a back-end. We give an overview of LLVM in Section 3.2. We then proceed to describe the compiler itself. We present its overall architecture in Section 3.3, its representation of supported non-process values in Section 3.4, its compilation of non-process expressions in Section 3.5, and its compilation of processes in Section 3.6. Finally, we evaluate its performance on a number of case studies in Section 3.7.

3.1 Motivation

FDR's eager evaluation of CSP_M allows processes to be compiled to efficient explicit GLTS or supercombinator representations. The explicit GLTS representation allows the *afters* (Definition 2.1.1) of a node to be computed with just a memory access. However, this comes with the upfront cost of pre-computing all states and transitions and with the runtime cost of storing them. It is a good trade-off where the transitions are visited frequently enough to justify caching them. In cases where only a small fraction of the transitions is visited during a check, much of this work could be avoided. Moreover, the existing evaluator interprets rather than compiles CSP_M. Due to the run-once nature of verification, this is often more efficient, but CSP_M's support for parameterised processes makes it simple to create short scripts that put a lot of pressure on the interpreter.

For example, consider the process $P(0)$ where:

$$P(i) = a \rightarrow P(i+1)$$
$$P(N) = a \rightarrow P(0)$$

for some large constant N, say $N = 10^9$. This would require evaluating $i + 1$ for N different values of i and creating a process environment with $N + 1$ distinct syntactic processes. Using such large values is not particularly common in explicit model checking due to the large state spaces involved, with techniques such as *data independence* [RB98] making it possible to prove certain results using smaller data types. However, using multiple parameters each with a small range can have the same result:

$$Q(b_1, ..., b_n) = \mathit{flip}.1 \rightarrow Q(\neg b_1, ..., b_n) \,\Box\, ... \,\Box\, \mathit{flip}.n \rightarrow Q(b_1, ..., \neg b_n)$$

Here, the process function Q represents 2^n distinct syntactic processes, one for each set of arguments. Since each of the Boolean parameters can be flipped independently, eagerly evaluating $Q(\mathit{false}, ..., \mathit{false})$ requires evaluating each of these processes. How-

ever, the set of states reachable in the overall system might be much smaller. Consider Q to represent a sequence of bits. We can then consider the system after no more than 3 bit flips as follows:

$$CosmicRay = flip?x \rightarrow STOP$$
$$System = Q(false, ..., false) \underset{\{|flip|\}}{\parallel} (CosmicRay \;|||\; CosmicRay \;|||\; CosmicRay)$$

In this combination, there are roughly n^3 states of Q reachable (all states with no more than 3 b_i set). For $n = 100$, 2^n states is intractable, but $n^3 = 10^6$ is quite manageable.

A particular class of problem where this is important is when modelling an intruder for model-checking cryptoprotocols. Such an intruder is parameterised by its knowledge of a number of facts, and is conceptually similar to Q above, where the known facts depend on the evolution of the protocol and not all combinations are reachable. The general intruder model in [RG97] uses a manual decomposition into multiple processes to obtain the desired lazy exploration behaviour by exploiting FDR's compilation strategy where parallel compositions are compiled to lazy supercombinator machines. The analysis of security protocols using such an intruder is discussed in detail in [RSG+01]. We consider a more straightforward implementation using lazy compilation in Section 3.7. Problems where such an insightful decomposition has not yet been discovered cannot be explored with FDR.

The motivation for introducing a lazy compiler in FDR is then mainly to allow the consideration of problems that were previously not supported and to efficiently support more natural representations of other problems. A lazy compiler is particularly well suited to *parameterised state machines*, that is, process functions with large argument spaces that use the sequential subset of CSP so that transitions can be computed efficiently. Besides supported new classes of problems, the use of a compiler rather than an interpreter can make evaluation quicker in some cases. We emphasise that the lazy compiler is not intended as a replacement for the existing evaluator, but rather as a choice to be used when it is more suitable.

If on average each component state is visited about once in the main check, then we need lazy compilation to be faster than the current interpretation to gain an advantage. On the other hand if each state is visited much less than once, then we can still gain a performance advantage even when compilation per state is slower.

In assessing the value of lazy compilation we might restrict ourselves purely to performance or might also pay regard to the possibility of choosing a natural parallel decomposition for a system rather than having to find one to make a check tractable. Good examples of this are the cryptoprotocol intruder, patience games such as peg solitaire, and state machines with a substantial parameter range such as those output by the industrial verification suite ASD [BH03].

3.1.1 Haskell back-end

We first investigated replacing the evaluator with a strict compiler, with the intention of improving evaluation performance as well as setting up a compilation framework that could support lazy compilation in a similar manner. This compiler was designed to output syntactic processes and feed into FDR's machine compiler (Section 2.2.2), whereas a lazy CSP_M compiler would have to bypass the machine compiler as well and feed into the state explorer since the machine compiler requires a fully evaluated syntactic process. Due to CSP_M's functional language's similarity to Haskell and the existing evaluator's Haskell-based implementation, we decided on Haskell as a back-end for this compiler and built a prototype implementation supporting most of CSP_M.[1] After a thorough evaluation of the prototype, we found its performance unsatisfactory, but the exercise was quite insightful and provided direction for the lazy compiler presented here. In particular, while the compiled code performed well, the compilation process took prohibitively long in all but the simplest cases.

Types and events. The main source of difficulty was mapping CSP_M's type system onto Haskell's. The *dot* operator commonly used to create structured events required

[1] Compilation of processes was implemented by the author, and compilation of non-process expressions as well as integration into FDR was implemented by Thomas Gibson-Robinson.

a complex framework of type classes, which pushed the Glasgow Haskell Compiler (GHC)[2] to its limits when compiling our generated code. In a typical example where evaluation using the `libcspm` evaluator would take 1-2 minutes, running the compiled generated Haskell code would take only 15-30 seconds, but the compilation would take several hours or run into compiler limits altogether.

Sets and complex data structures. Even disregarding the compilation time (which could conceivably be reduced with compiler hints or amortised by compiling portions of the code ahead of time and reusing them), we did not find much performance improvement when dealing with sets or other complex data structures. This is not surprising, since operations on these structures are more costly than the interpretation overhead and there is not much more room for optimising their compiled variants than optimising the relevant parts of the interpreter.

Parameterised processes. As expected, in instances where the compiled code performed better than the interpreter, the improvement was more marked the more times this code would run. In particular, non-process functions called many times would show such an improvement. Unfortunately, due to the strict compilation, each invocation of a process function still had to be added to an environment, which could not be optimised by a compiler, and which was typically expensive enough to make the benefits of its compilation negligible.

We took the above findings into consideration when designing our lazy compiler. Having noted that many of CSP$_M$'s more advanced features did not benefit from compilation but added to the complexity of the compiler and negatively impacted the compilation time, we decided to make a streamlined compiler for the subset of CSP$_M$ that would benefit most from compilation and allow it to be used in a single script alongside the interpreted evaluator to support the full generality of CSP$_M$. To avoid clashing with our target's type system, we took advantage of the fact that the input

[2] `https://www.haskell.org/ghc/`

to the compiler has already been type-checked and targeted the low-level LLVM infrastructure, working with a stream of bits and coercing to and from the necessary types without relying on a secondary type checker (Section 3.4). We focused on supporting parameterised state machines and designed the compiler to represent them in a structured form, bypassing the syntactic process environment and the machine compiler.

3.2 LLVM

The LLVM Compiler Infrastructure [LA04, Lat02] is a language- and platform-independent multi-stage compilation and optimisation framework. Compiler implementers can target a type-safe universal intermediate representation specified by LLVM. LLVM provides a number of analysis, transformation, and code generation passes that can be used to transform the input into optimised native code for a number of supported platforms. All of these passes can be performed ahead of time producing a traditional compiled binary or at run time, using an in-memory linker. All of these features make it an attractive target for a cross-platform compiler. The support for efficient just-in-time compilation in particular is useful for verification tools such as FDR since its scripts are not typically run multiple times.

Users have a choice of three representations for LLVM code: an in-memory representation generated using a C or C++ API, a compact binary format called LLVM Bitcode, and an assembly-like, human-readable format called LLVM Intermediate Representation (IR). We will focus on the latter format since it is best suited for presentation.

The lazy compiler emits IR in this textual format. This choice was made to simplify implementation given the particulars of FDR and may not be optimal for other implementations: among other considerations, it incurs an overhead from eventually parsing the IR. However, in our testing, the emission and parsing of the IR took a negligible amount of time.

```
declare void @crash() noreturn ; crashes to signal an error condition

define i32 @factorial(i32 %n) {
  %negativeArg = icmp slt i32 %n, 0 ; negative arguments not valid
  %largeArg = icmp sge i32 %n, 13 ; 13! can't fit in an i32
  %badArg = or i1 %negativeArg, %largeArg
  br %badArg, label %badExit, label %body
badExit:
  call void @crash() noreturn
  unreachable ; unreachable since the call to @crash does not return normally
body:
  ; allocate a location in memory so the result can be stored from both branches
  %resultPtr = alloca i32, i32 1
  %isZero = icmp eq i32 %n, 0
  br i1 %isZero, label %baseCase, label %recurse
baseCase: ; n=0, 0! = 1
  store i32 1, i32* resultPtr
  br successExit
recurse: ; n ≠ 0, n! = (n-1)! * n
  %0 = sub i32 %n, 1 ; n-1
  %1 = call i32 @factorial(i32 %0) ; (n-1)!
  %2 = mul i32 %1, %n ; (n-1)! * n
  store i32 %2, i32* resultPtr
  br successExit
successExit:
  %result = load i32, i32* resultPtr
  ret i32 %result
}
```

Figure 3.1: An implementation of the factorial function in LLVM.

In this section, we will give an overview of the LLVM language, focusing especially on the features we used in the lazy compiler. More details are available online in the official LLVM Language Reference Manual[3]. We will refer to the factorial function in Figure 3.1 as a running example throughout this section. This takes a signed 32-bit integer between 0 and 12 inclusive and returns its factorial.

LLVM organises code into modules, which consist of functions, global variables, and type declarations, as well as some more esoteric features. We will only cover functions and type declarations here since our compiler does not make use of global variables (FDR's multi-core and cluster functionality make global state highly undesirable). A simple LLVM function declaration looks like:

[3] `http://llvm.org/docs/LangRef.html`

```
declare %return_type @name(%arg1_type %arg1_name, %arg2_type %arg2_name, ...)
```

The `declare` keyword can be followed by `private` to indicate that the function will only be used in this module, allowing for more aggressive, ABI-breaking optimisations[4]. A function definition is similar, however the `define` keyword is used instead of `declare` and a body is provided. In Figure 3.1, we declare a `@crash` function and define a `@factorial` function.

3.2.1 LLVM Type System

LLVM uses a strong type system, which allows for a number of optimisations and analyses that might not otherwise be feasible. We list here a selection of types relevant to the lazy compiler.

Fixed-width integers. LLVM integers must have a specified fixed bit width, although this width is arbitrary and effectively unbounded. These are specified as type `i`N, where N is an integer between 1 to about 8 million. For example, `i32` is a 32-bit integer. These integers are simply a sequence of bits; it is up to individual instructions to interpret them as signed or unsigned. Boolean variables are usually represented as 1-bit integers `i1`.

Void. The `void` type cannot hold any values. It is typically used as the nominal return type for functions without a return value.

Functions. Function types use the same syntax as in C: `return(arg1, arg2)` is the type of a function that takes arguments of types `arg1` and `arg2` and returns a value

[4]Platforms typically define an *application binary interface* (ABI), which allows functions compiled separately to interface with each other. This is a set of conventions indicating, among other things, where in memory or in registers certain arguments need to be passed. If a function is not exposed to the outside world, it doesn't need to comply with the ABI. This provides the compiler more opportunities for optimisation, such as rearranging the function's arguments or removing some of its arguments altogether.

of type `return`. For example, an integer square root function on 32-bit integers might have the type `i16(i32)` and a less-than function might have the type `i1(i32, i32)`.

Pointers. A pointer to a memory location holding an object of type `ty` is written as `ty*`. For example a pointer to a 32-bit integer would be `i32*`, a pointer to a peek function that reads and returns the 32-bit integer at a specified memory location would be `i32(i32*)*`. Unlike C, LLVM does not support a `void*` type. Instead, `i8*` is used to represent memory locations of unspecified type.

Arrays. An array type `[`N `x T]` is the type of a contiguous sequence of N objects of type `T`. For example, a string of ten 8-bit characters would have the type `[10 x i8]`.

Structures. LLVM structures are records consisting of an arbitrary number of unnamed fields. They can be defined in line by specifying its component types and they can be named at the top level of a module using a type declaration. A structure type looks like `{field1, field2, ...}`. For example, a nullable integer can be represented by `{i1, i32}`, with the first field indicating whether the integer is null, and the second field holding the value. A type declaration looks like `%name = type {fields}`. For example, `%int_triple = type {i32, i32, i32}`.

3.2.2 LLVM Instructions

An LLVM function consists of at least one *basic block*, which is a sequence of regular instructions followed by a single *terminator* instruction. Within a basic block, execution is sequential while terminators control the flow between basic blocks.

LLVM uses *static single assignment* (SSA) form, meaning that each variable is assigned exactly once. This simplifies analysis by making use-define chains explicit; however, most interesting problems require using conditional control flow to set a single variable to one of a number of values. This typically requires the use of a special Φ function to *merge* distinct variables into one depending on control flow incoming to a

basic block. LLVM provides this in the form of the `phi` instruction, described in more detail below.

In addition to SSA variables, LLVM supports accessing memory through typed pointers. Memory locations are not subject to the single-assignment restriction, but they can only be accessed by dedicated memory instructions; arithmetic instructions, for example, only work on variables. Memory can be used to avoid the need for Φ instructions by storing live variables at the end of each basic block and loading them at the start. This need not incur any run-time performance penalty since LLVM has built in transformation passes to convert between the memory-using and the Φ-using forms. The example in Figure 3.1 uses `%resultPtr` this way; to avoid the allocation, the `store`s in line 17 and line 23 could instead assign to distinct variables, and the `load` in line 26 could be replaced by a `phi` instruction.

The rest of this section will list a selection of LLVM instructions emitted by the lazy compiler, starting with memory instructions, followed by regular instructions, and concluding with terminator instructions. Most of the described instructions can take additional markers or arguments, but as these are not necessary for understanding the rest of this chapter, we will not mention them. A typical instruction has the form `%result = instr ty0 %arg0, ty1 %arg1`, applying `instr` to arguments `%arg0` and `%arg1` of types `ty0` and `ty1` respectively. The type of `%result` is typically determined by the instruction or specified as an additional argument.

Alloca, Load, Store. The instruction `%ptr = alloca ty, i32` *N* allocates space for *N* elements of type `ty` on the stack and places the result, of type `ty*` in `%ptr`. To write a value `%val` of type `ty` through a pointer `%dst`, the `store ty %val, ty* %dst` instruction is used. Conversely, `%val = load ty, ty* %src` loads a value of type `ty` from a pointer `src` into `val`. These instructions are used in Figure 3.1 in lines 13, 17, 23, and 26.

Undef. The `undef` literal represents a constant of specified type whose value is not yet known. This can be useful for declaring a value of a structure type and initialising it one member at a time using the `insertvalue` instruction as below.

InsertValue, ExtractValue. These instructions write and read aggregate (structure or array) fields by index. Note that since aggregates are first-class types in LLVM, these instructions do not access memory. Consequently, `insertvalue` does not modify its aggregate argument in place but returns a modified copy of it; despite these notional semantics, the code generator will generally modify the value in place and not make the potentially expensive copy unless it is required. To write a value `%val` of type `valty` to an aggregate `%dst` of type `dstty` at 0-based index N, we can use the syntax `%result = insertvalue dstty %dst, valty %val,` N, where `%result` will have the type `dstty`. To read the value at index N from an aggregate `%src` of type `srcty`, we use `%val = extractvalue srcty %src,` N; the type of `%val` is statically inferred.

```
1 %pair1 = insertvalue {i1, i32} undef, i1 1, 0  ; {i1 1, i32 undef}
2 %pair2 = insertvalue {i1, i32} %pair2, i32 1234, 1 ; {i1 1, i32 1234}
3 %val1 = extractvalue {i1, i32} %pair2, 0       ; i1 1
4 %val2 = extractvalue {i1, i32} %pair2, 1       ; i32 1234
```

Call. A function `@fn` of type `retty(argty0, argty1)` can be called with arguments `%arg0` and `%arg1` as follows: `%result = call retty @fn(argty0 %arg0, argty1 %arg1)`.

ICmp. The `icmp` *cc* `ty %arg0, %arg1` instruction compares two integers `%arg0` and `%arg1`, both of type `ty` and returns a Boolean according to the condition code *cc*. Valid condition codes are: `eq` for equality, `ne` for non-equality, `ugt` for unsigned greater-than, `uge` for unsigned greater-than-or-equal, `ult` for unsigned less-than, `ule` for unsigned less-than-or-equal, and `sgt`, `sge`, `slt`, and `sle` for the signed versions of the above. Examples can be seen in Figure 3.1 in lines 4, 5, and 14.

ZExt, SExt, Trunc. These instructions change the bit width of an integer argument `%val` from `i`N to `i`M. Where $M > N$, `zext i`N `%val to i`M performs zero-extension and `sext i`N `%val to i`M performs sign-extension. Where $M < N$, `trunc i`N `%val to i`M truncates the argument.

Arithmetic and logic operations. We will not describe the simple binary arithmetic and logic instructions in detail here as their meanings should be clear from the mnemonics or else readily available in the LLVM Language Reference Manual. They take the form `%result = instr ty %arg0, %arg1`, with both arguments and the result sharing the type `ty`. For example, `%sum = add i32 %a, %b` adds the 32-bit integers *a* and *b* and puts the result in *sum*, and `%off = not i1 %on` places the negation of a Boolean flag *on* into *off*. There are a number of examples in Figure 3.1.

As mentioned above, LLVM does not have separate signed and unsigned types. For some instructions (such as `add` and `sub`), the signedness of the arguments does not matter. For other operations, such as division, there are separate unsigned and signed versions: `udiv` performs unsigned division and `sdiv` performs signed division.

Ret. The `ret` terminator instruction returns control flow and a return value (for non-void functions) to the caller. It is used as `ret ty %val` or `ret void`. This is shown in line 27 of Figure 3.1.

Br. The `br` terminator can be used to implement conditional or unconditional branches. The unconditional form looks like `br label %dst` and transfers control flow to the basic block labelled `dst`. The conditional form `br i1 %cond, label %then, label %else` transfers control to `then` if `%cond` is true (1) and to `else` otherwise. We can see the conditional form in line 7 and line 15 of Figure 3.1 and the unconditional form in line 18 and line 24.

Switch. The `switch` terminator instruction is similar to a conditional branch but operating on an integer instead of a Boolean. It takes a list of value-target pairs as well as a *default* target for when the value does not match any of the values in the list as follows.

```
switch iN %value, label %default [
    iN v0, label %dst0 ; if %value = v0, go to dst0
    iN v1, label %dst1 ; if %value = v1, go to dst1
    ...
    iN vn, label %dstn ; if %value = vn, go to dstn
] ; else, go to default
```

Phi. This is a special instruction used to reconcile SSA's requirement to assign to each variable exactly once (syntactically) with the desire to use control flow to assign different values to the same variable. To use it, each of the relevant basic blocks assigns the desired value to its own "version" of the variable. The basic block where control flow is merged uses the `phi` instruction with a list of these variables and the basic blocks where they are defined. The result of the `phi` instruction is the variable corresponding to the predecessor basic block that just executed. For example, the following C code

```
if (flag)
    x = a + 1;
else
    x = a + 2;
y = x;
```

could be translated as follows:

```
br i1 %flag, label %then, label %else
then:
  %x0 = add i32 %a, 1
  br %endif
else:
  %x1 = add i32 %a, 2
  br %endif
endif:
  %y = phi i32 [ %x0, %then ], [ %x1, %else ]
```

Unreachable. This is a terminator instruction that is assumed by the optimiser to not be reachable. It can be used, for example, for the default branch of a `switch` instruction to indicate that the inputs will always be covered by one of the specific branches, or as in line 10 of Figure 3.1 to indicate that a preceding function call cannot return.

3.3 Overview

Our non-invasive approach to adding a lazy compiler to FDR3 allows most of a CSP$_M$ script to flow through FDR3 largely as described in Section 2.2. However, a special `lazy_compile` operator causes its parsed and type-checked argument to bypass the evaluator and proceed as far as the strategy executor unchanged[5], as shown in Figure 3.2. The lazy compiler then translates it to LLVM IR, which is then compiled and optimised by LLVM. The interface between FDR3's refinement checker and the generated LLVM code is provided by a dedicated *lazy-compiled* machine type, separate from the explicit and supercombinator machines described in Section 2.2.2. Finally, this machine can be *lazy-enumerated*[6] to provide an interface similar to explicit machines and allow it to be used as a component of a supercombinator machine.

A *lazy-compiled* machine is initialised with an *event map* mapping the *structured* events produced by the lazy-compiled code to the *compiled*, integer, events expected by FDR3. It wraps the RootNode (see Figure 3.3) and VisitTransitions (see Figure 3.4) functions exported from the lazy-compiled code, marshalling the nodes and converting from the visitor-based transition interface to an iterator-based one. Note, however, that the figures only outline the contract of the functions rather than their implementation; in particular, VisitTransitions does not have access to an *afters* function, and indeed the bulk of the lazy compiler is tasked with generating the *afters*. The event representation used by VisitTransitions is described in detail

[5]This was implemented by Thomas Gibson-Robinson.

[6]The author's implementation of lazy enumeration for lazy-compiled machines extends FDR3's existing lazy enumeration operator.

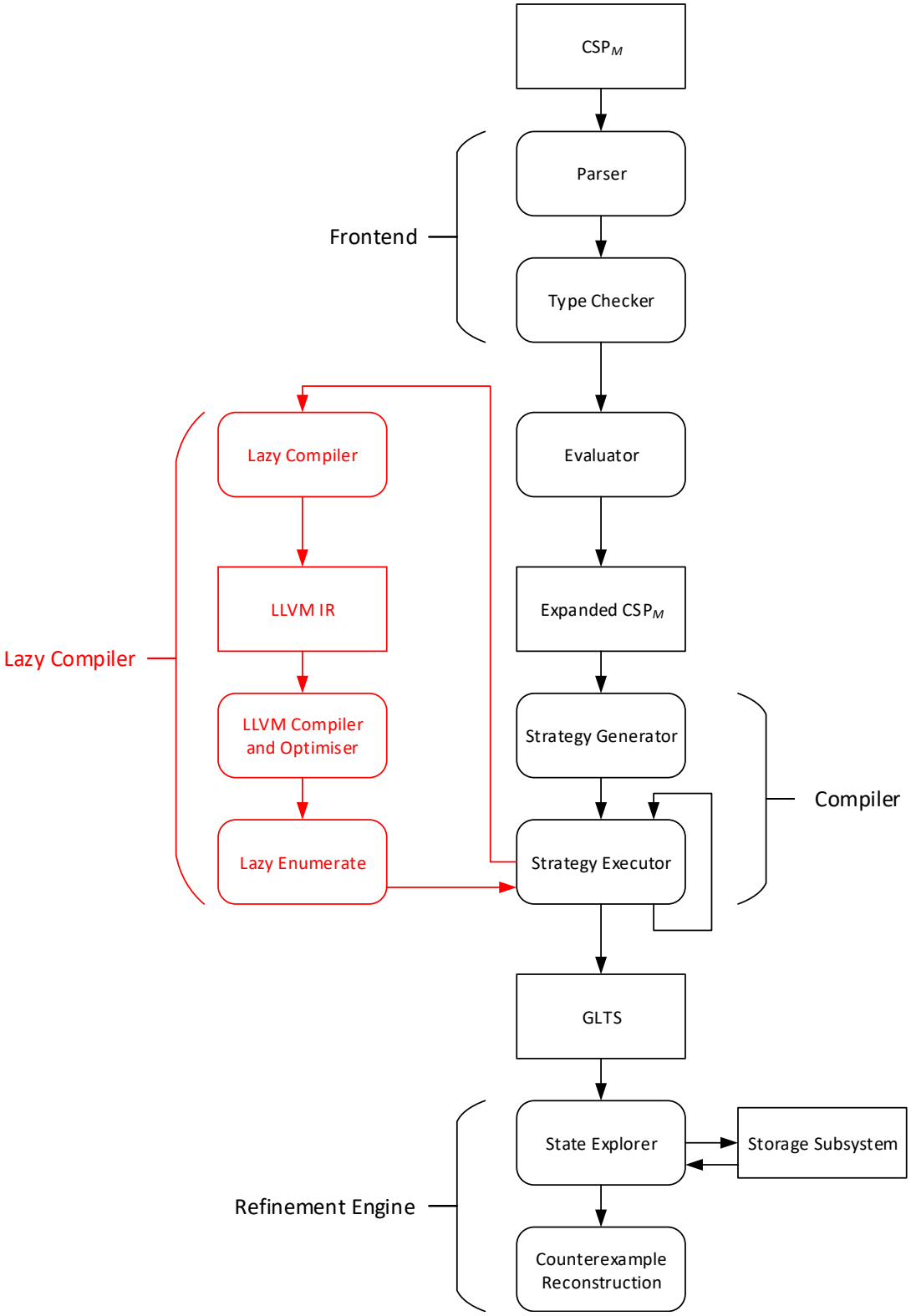


Figure 3.2: FDR3 data flow with lazy compilation.

```
1: function RootNode(byref buffer)
2:     buffer ← ⟨root node⟩
3: end function
```

Figure 3.3: RootNode simply writes a representation of the root node into the provided buffer.

```
1: function VisitTransitions(node, context, visitor)
2:     for each (event, after) ∈ afters(node) do
3:         visitor(context, event, after)
4:     end for
5: end function
```

Figure 3.4: VisitTransitions calls the provided visitor with the provided context for each *after* of the provided node.

in Section 3.4 and the node representation used by both functions is described in Section 3.6.

CSP_M can be broadly divided into classical CSP process expressions and a functional language inspired by the notation used in "blackboard" CSP, augmenting processes with arguments and providing users with a mechanism for computing these arguments at run time. Section 3.5 covers the compilation of much of the supported functional language (focusing particularly on non-process expressions) and Section 3.6 covers the compilation of process expressions.

3.4 Type Mapping

Of the types supported by CSP_M, the lazy compiler has full support for Boolean values, 32-bit integers, and non-recursive user-defined algebraic datatypes containing only these types, as well as limited support for finite sequences of these types. It also has minimal support for the types of events and processes, as necessary to communicate with the rest of FDR. For each of these types, there is both a *working* representation used within the LLVM code and a *packed* representation that is passed on to FDR and persists for the duration of the session.

In the working representation, Booleans and 32-bit integers are stored as the corresponding LLVM types `i1` and `i32`, respectively. Algebraic datatypes are conceptually

```
; datatype X = C_0.A_0.A_1.....A_{n_a} | C_1.B_0.B_1.....B_{n_b} | ... | C_{m-1}.Z_0.Z_1.....Z_{n_z}

%X = type { iN, ; Tag indicating the constructor
    { %A_0, %A_1, ..., %A_{n_a} }, ; Valid whenever the tag is 0; C_0.A_0.A_1.....A_{n_a}
    { %B_0, %B_1, ..., %B_{n_b} }, ; Valid whenever the tag is 1; C_1.B_0.B_1.....B_{n_b}
    ...,
    { %Z_0, %Z_1, ..., %Z_{n_z} } ; Valid whenever the tag is m-1; C_{m-1}.Z_0.Z_1.....Z_{n_z}
}
```

Figure 3.5: Skeleton for working representation of datatypes.

discriminated unions, and we would like to store them as such. LLVM does not have built-in support for union types, but it does allow type punning, and a typical approach is to declare the union as a byte array at least as long as the longest member and to cast it to the necessary type on each access. For our prototype, we decided to use a structure (record) type instead. This simplifies the implementation by removing the type casts and the need to calculate and keep track of the size (including padding) of the components. It wastes space, but that is not a problem since the working representation is used only briefly, when computing a state's *afters*. More precisely, a datatype is converted to a structure whose first member is a tag (starting at 0) indicating the constructor used and whose following members are structures corresponding to each of the alternatives, with each structure containing a member corresponding to each of the components of the alternative. This is illustrated in the general case in Figure 3.5, which assumes that the component types such as A_0 are already translated as $\%A_0$, etc. The tag type is an integer of the minimum width required to hold all possible values – namely $\lceil \log_2 m \rceil$, where m is the number of constructors (with the caveat that LLVM does not support an `i0` type, so for datatypes with only one constructor, we still have to use an `i1` as a tag). An example is shown in Figure 3.6.

The packed representation is a stream of bits, grouped into aligned 64-bit chunks (for efficient access on 64-bit platforms). Integers and Booleans are streamed without modification. For datatypes, first the tag is streamed, followed recursively by each component of the corresponding union member. If any value would cross a chunk boundary, padding is inserted into the first chunk and the value is placed entirely

```
; datatype T = Wrapper.Bool
%T = type { i1, ; Tag (0..0); uses 0 bits in the packed representation
    { i1 } ; Wrapper.Bool
}

; datatype U = Empty | Full.Int.Bool | Wrapped.T
%U = type { i2, ; Tag (0..2)
    { }, ; Empty
    { i32, i1 }, ; Full.Int.Bool
    { %T } ; Wrapped.T
}
```

Figure 3.6: Example working representation of datatypes.

in the second. Note that though this means atomic values longer than 64 bits are not supported (as in FDR's current compiler), composite types such as sequences are serialised component-wise, so the restriction only applies to their components. A more efficient packing algorithm could be used, but the amount of space needed depends on the contents of the particular instance of a datatype (e.g., the type `U` in Figure 3.6 could require 2 bits for `Empty`, 35 bits for `Full.0.False`, or 3 bits for `Wrapped.Wrapper.False`) rather than on its static type, so the packing would have to be done at run time. This excludes many typical algorithms due to performance considerations, and we have found the overhead of the present approach to be acceptable.

Events are similar to datatypes, but with enough differences to need special treatment. Most importantly, while well-behaved CSP$_M$ scripts do not use partially constructed datatypes, they do frequently use partially constructed events, for example implicitly in the context of a prefix input. Additionally, the set of valid events is aggregated from all of the `channel` declarations in a CSP$_M$ script (as well as the implicitly declared τ and $\checkmark$ events). This means that treating them the same way as datatypes would disallow the use of lazy compilation within any scripts that use unsupported events, even if those events are not referenced in the lazy-compiled areas. Finally, events are most often *performed* (with a prefix), rather than compared or modified like other data types. This allows us to forgo a working representation, and only provide a packed one. The lack of a working representation disallows the use of events anywhere except prefix, allowing us to statically determine the constructors that can be used

to construct them within a given process. We then synthesise a datatype using these constructors and use the same packed representation as for a normal datatype. Due to the streaming nature of this representation, partially constructed events are easy to work with, and due to the possibility of static analysis, unused channels do not need to be translated to LLVM and can contain arbitrary CSP_M types.

For the sake of completeness, we will include a brief outline of the representation of processes, although they are described in more detail in Section 3.6. Processes are conceptually represented as a *format identifier* along with the values of each free variable in the corresponding format. The packed representation is a bit stream similar to that used to store datatypes. The in-memory representation uses an LLVM register for each of the bindings, but does not store the format explicitly; instead the format is used to select the branch of code that will be executed.

3.4.1 Serialisation Functions

To facilitate conversion between the working and packed representations, we generate a number of serialisation and deserialisation functions. These include hard-coded functions to serialise and deserialise arbitrary-width integers (up to 64 bits, since our format uses 64-bit chunks) and a pair of functions generated for each datatype.

Each serialisation function takes the value to serialise and a destination pointer containing the address of the buffer and the offset in bits within the buffer. The integer serialisation function additionally takes the width in bits of the value to be serialised; we will not discuss it in detail here as its implementation is straightforward. Each datatype serialisation function uses the constructor tag to dispatch control to a dedicated branch for that constructor. Each branch serialises each value in the constructor in sequence. Figure 3.7 shows the skeleton for the datatype X in Figure 3.5.

Similarly, the deserialisation functions take a bit pointer to read from and return a new bit pointer (from which the next value may be read) and the value read, with the integer deserialisation function additionally taking the bit width of the integer to read. To deserialise a datatype, we first deserialise its constructor tag and then dispatch

```
; datatype X = C_0.A_0.A_1.....A_{n_a} | C_1.B_0.B_1.....B_{n_b} | ... | C_{m-1}.Z_0.Z_1.....Z_{n_z}

%bitptr = type {i64*, i32} ; Pointer to buffer and offset in bits

; Serialises %value to %dest using %width bits.
; Returns the appropriately incremented pointer.
declare private %bitptr @ser_int(%bitptr %dest, i8 %width, i64 %value)

define private %bitptr @"ser!X"(%bitptr %dest, %X %value) {
entry:
    %tag = extractvalue %X %value, 0 ; Extract the tag
    ; Serialise the N-bit tag
    %tag64 = zext iN %tag to i64
    %dest.tag = call %bitptr @ser_int(%bitptr %dest, i8 N, i64 %tag)

    switch iN %tag, label %default [ ; Dispatch to branch based on tag
        iN 0, label %C_0
        iN 1, label %C_1
        ...
        iN m-1, label %C_{m-1}
    ]
C_0:
    %Avals = extractvalue %X %value, 1 ; A_0.A_1.....A_{n_a}
    ; Extract and serialise each component
    %A_0val = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %Avals, 0
    %dest.A_0 = call %bitptr @"ser!A_0"(%bitptr %dest.tag, %A_0 %A_0val)
    %A_1val = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %Avals, 1
    %dest.A_1 = call %bitptr @"ser!A_1"(%bitptr %dest.A_0, %A_1 %A_1val)
    ...
    %A_{n_a}val = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %Avals, n_a
    %dest.A_{n_a} = call %bitptr @"ser!A_{n_a}"(%bitptr %dest.A_{n_a-1}, %A_{n_a} %A_{n_a}val)
    ret %bitptr %dest.A_{n_a} ; Return the final pointer
; Similarly for C_1 ... C_{m-1}
default:
    unreachable ; There is a branch for every valid tag
}
```

Figure 3.7: Skeleton serialisation function.

control to a dedicated branch, where each value in that constructor is deserialised in sequence. This is illustrated in Figure 3.8 for the generic datatype X.

3.4.2 Sequences

In existing work, sequence-like objects are usually represented as linked lists, contiguous arrays, or a combination of the two. For finite sequences, the length is either stored alongside the data or indicated implicitly with a dedicated *null terminator* object. Compared to arrays, linked lists have the advantage of quick insertions and deletions. Arrays, however, require simpler memory management and provide faster random access, both desirable for the present CSP$_M$ compiler. In particular, the working representation is temporary and the packed representation is immutable and intended to be trivially copyable. With this in mind, we have chosen a length coupled with a pointer to a stack-allocated array for the working representation of sequences. That is, for $\langle T \rangle$, the working representation would be `{i32, %T*}`. The pointer member would actually point to an appropriately sized array, namely, for a sequence of length n, it would point to an object of type `[`n `x %T]`.

The packed representation is a bit stream as for datatypes, with the length streamed first, followed by each of the elements. Unlike integral or user-defined types, sequences are serialised in line rather than by dedicated functions. This is desirable since determining the sequence types used by a CSP$_M$ script is not always trivial, and without some sort of static analysis, the number of sequence types would be unbounded. (Even a script that does not define any datatypes could still use $\langle Int \rangle$, $\langle\langle Int \rangle\rangle$, $\langle\langle\langle Int \rangle\rangle\rangle$, etc.)

```
; datatype X = C_0.A_0.A_1.....A_{n_a} | C_1.B_0.B_1.....B_{n_b} | ... | C_{m-1}.Z_0.Z_1.....Z_{n_z}

; Deserialises a value of width %width bits from %src.
; Returns the appropriately incremented pointer and the value.
declare private {%bitptr, i64} @deser_int(%bitptr %src, i8 %width)

define private {%bitptr, %X} @"deser!X"(%bitptr %src) {
entry:
    ; Deserialise the N-bit tag.
    ; For brevity, we'll use an (invalid) shorthand for the following:
    ; %retval = call {%bitptr, i64} @deser_int(%bitptr %src, i8 N)
    ; %src.tag = extractvalue {%bitptr, i64} %retval, 0
    ; %tag64 = extractvalue {%bitptr, i64} %retval, 1
    {%src.tag, %tag64} = call {%bitptr, i64} @deser_int(%bitptr %src, i8 N)
    %tag = trunc i64 %tag64 to iN
    ; Write the tag into an unitialised %X value.
    %result.tag = insertvalue %X undef, iN %tag, 0

    switch iN %tag, label %default [ ; Dispatch to branch based on tag
        iN 0, label %C_0
        iN 1, label %C_1
        ...
        iN m - 1, label %C_{m-1}
    ]
C_0:
    ; Deserialise each component
    {%src.A_0, %A_0val} = call {%bitptr, %A_0} @"deser!A_0"(%bitptr %src.tag)
    %Avals.A_0 = insertvalue { %A_0, %A_1, ..., %A_{n_a} } undef, %A_0val, 0
    {%src.A_1, %A_1val} = call {%bitptr, %A_1} @"deser!A_1"(%bitptr %src.A_0)
    %Avals.A_1 = insertvalue { %A_0, %A_1, ..., %A_{n_a} } %Avals.A_0, %A_1val, 1
    ...
    {%src.A_{n_a}, %A_{n_a}val} = call {%bitptr, %A_{n_a}} @"deser!A_1"(%bitptr %src.A_{n_a-1})
    %Avals.A_{n_a} = insertvalue { %A_0, %A_1, ..., %A_{n_a} } %Avals.A_{n_a-1}, %A_{n_a}val, n_a
    ; Insert the now deserialised A_0.A_1.....A_{n_a} fields into the result structure
    %result = insertvalue %result.tag, { %A_0, %A_1, ..., %A_{n_a} } %Avals.A_{n_a}, 1
    ret {%bitptr, %X} {%src.A_{n_a}, %result} ; Return new pointer and value
; Similarly for C_1 ... C_{m-1}
default:
    unreachable ; There is a branch for every valid tag
}
```

Figure 3.8: Skeleton deserialisation function.

3.5 Compiling Non-Process Expressions

Since the compilation of integer and Boolean expressions, as well as conditional expressions on first-class objects, is already covered in depth in many compiler textbooks and tutorials, we will not cover it again here. The example in Figure 3.9 illustrates the translation of the simple expression `flag && a + 2 < b` to LLVM IR.

Instead, this section will cover the implementation of datatypes (Section 3.5.1), patterns (Section 3.5.2), statements (Section 3.5.3), sequences (Section 3.5.4), and functions (Section 3.5.5). Section 3.5.4 will additionally cover the implementation of sets, which are not first-class citizens in the lazy compiler but nevertheless appear in prefix and replicated external choice expressions, where they are produced and immediately consumed without being materialised.

3.5.1 User-Defined Datatypes

The three main operations on user-defined datatypes supported by the lazy compiler are construction, decomposition, and equality testing. Construction with the dot operator is straightforward, with the interesting details already covered in Section 3.4. Decomposition is supported by pattern matching and is covered in Section 3.5.2. This section will therefore only cover equality testing.

Similarly to the serialisation and deserialisation functions, we generate an equality function for each user-defined datatype. Each function takes two values of the corresponding datatype and first compares their tags. If they are the same, it dispatches to a branch corresponding to that tag and recursively compares the fields of that constructor. Otherwise, it simply returns `false`. The skeleton of this function for a datatype X (as in Figure 3.5) is illustrated in Figure 3.10.

```
1 %0 = add i32 %a, 2 ; a + 2
2 %1 = icmp slt i32 %0, %b ; a + 2 < b
3 %2 = and i1 %flag, %1 ; flag && a + 2 < b
```

Figure 3.9: Translation of a simple expression to LLVM.

```
; datatype X = C_0.A_0.A_1.....A_{n_a} | C_1.B_0.B_1.....B_{n_b} | ... | C_{m-1}.Z_0.Z_1.....Z_{n_z}

define private i1 @"eq!X"(%X %left, %X %right) {
entry:
    %tagL = extractvalue %X %left, 0 ; Extract the tags
    %tagR = extractvalue %X %right, 0
    %same_tag = icmp eq iN %tagL, %tagR
    br i1 %same_tag, label %dispatch, label %not_equal

not_equal:
    ret i1 0 ; Return false

dispatch:
    switch iN %tagL, label %default [ ; Dispatch to branch based on tag
        iN 0, label %C_0
        iN 1, label %C_1
        ...
        iN m - 1, label %C_{m-1}
    ]
C_0:
    %AvalsL = extractvalue %X %left, 1 ; A_0.A_1.....A_{n_a}
    %AvalsR = extractvalue %X %right, 1
    ; Extract and compare each component
    %A_0valL = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %AvalsL, 0
    %A_0valR = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %AvalsR, 0
    %A_0equal = call i1 @"eq!A_0"(%A_0 %A_0valL, %A_0 %A_0valR)
    ...
    %A_{n_a}valL = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %AvalsL, n_a
    %A_{n_a}valR = extractvalue { %A_0, %A_1, ..., %A_{n_a} } %AvalsR, n_a
    %A_{n_a}equal = call i1 @"eq!A_{n_a}"(%A_{n_a} %A_{n_a}valL, %A_{n_a} %A_{n_a}valR)
    ; AND the results of all the comparisons
    %r1 = and i1 %A_0equal, %A_1equal
    %r2 = and i1 %r1, %A_2equal
    ...
    %r{n_a} = and i1 %r(n_a - 1), %A_{n_a}equal
    ret i1 %r{n_a}
; Similarly for C_1 ... C_{m-1}
default:
    unreachable ; There is a branch for every valid tag
}
```

Figure 3.10: Skeleton datatype equality function.

3.5.2 Patterns

As is typical of functional languages, CSP$_M$ favours *pattern binding* expressions over simple assignment. Our prototype of the lazy compiler supports *wildcard*, *variable*, and *dot* patterns where the first clause is a datatype constructor. The wildcard pattern `_` matches any value and ignores it. A variable pattern `v` where `v` is a channel or datatype constructor matches that channel or constructor and does not bind any values. Otherwise, the variable pattern matches any value and adds a corresponding binding to the environment. Dot patterns like `a.b` are the primary way to decompose datatype values; they allow recursively matching the fields of a datatype to a sequence of patterns and add the corresponding set of bindings to the environment. Patterns are used to bind the left-hand side of a generator statement (see Section 3.5.3), to bind function arguments (see Section 3.5.5), and to match the input fields of a prefix expression (see Section 3.6.2).

For example, given a linked list datatype like the following:

```
datatype List = Nil | Cons.Int.List
```

we can write the following functions:

```
list_empty(l) = l == Nil
list_head(Cons.car._) = car
list_2nd(Cons._.(Cons.cadr._)) = cadr
```

The `list_empty` function uses a straightforward variable pattern to bind its argument to a name. The `list_head` function needs to decompose its argument into the head and tail of a list and extract the head. Accordingly, it uses a dot pattern to match the corresponding clause of `List`. The `Cons` variable pattern selects the clause that is to be matched, `car` binds the list head to a variable, and the wildcard binding discards the unneeded tail. The `list_2nd` function that extracts the second element of a `List` illustrates the use of nested dot patterns. In it, the head is discarded and the tail is matched against the same pattern as in `list_head` to extract the head of the tail.

3.5.3 Statements

CSP_M has a number of comprehension constructs for generating new sets or sequences based on existing ones. These include *list comprehensions* (see Section 3.5.4) as well as *replicated* process operators (see Section 3.6.2). These comprehension constructs use *statements*, which generate and restrict a sequence of values, and then consume them in a construct-specific way. For example,

$$\langle x \mid x \leftarrow \langle 1..10 \rangle,\ x \,\%\, 3 \neq 0 \rangle$$

constructs a sequence of integers between 1 and 10 not divisible by 3 and

$$\Box\ i \leftarrow \{1..3\}\ @\ out.i \rightarrow STOP$$

provides an external choice of $out.1$, $out.2$, and $out.3$. In the first example, there is a sequence statement and a predicate statement to the right of the |, and in the second example, there is a set statement before the @.

Given a consumer *Consumer* and a sequence of statements $s_1, s_2, ..., s_n$, we process the statements left to right, augmenting the environment with new bindings if necessary, and feed the eventual output into *Consumer*. Conceptually, the compiler performs a right fold on the sequence of statements using *Consumer* as the initial value. This is illustrated in Figure 3.11.

The two types of statements are *generators* and *predicates*. Predicates are simple Boolean expressions that restrict the produced values to those for which the predicate e is true. These are compiled to simple conditional statements, as shown in Figure 3.11. Generators of the form $p \leftarrow e$ produce each value in the set or sequence e and bind it to pattern p for consumption by subsequent statements and the consumer. These are compiled to for-loops, though how specifically this is accomplished depends on the expression and its type, as illustrated in Figure 3.12.

If e can be compiled to a sequence, we compile it as such and emit an indexed

CompileStatements(*Consumer*, *statements*) =
    foldr(CompileStatement, *Consumer*, *statements*)
function CompileStatement(*statement*, *body*)
    if statement **is** predicate e **then**
        Emit code: `if e do { body }`
    else if statement **is** generator $p \leftarrow e$ **then**
        Emit code: `foreach x in e do { p ← x; body }`
    end if
end function

(a) An outline of the algorithm that compiles statements.

```
// CompileStatements(Body, ⟨p1 ← e1, e2, p3 ← e3⟩)
foreach x1 in e1 do: // p1 ← e1
    p1 ← x1
    if e2 do: // e2
        foreach x3 in e3 do: // p3 ← e3
            p3 ← x3
            Body
```

(b) A skeleton of the emitted code.

Figure 3.11: An overview of how statements are compiled.

for-loop that iterates over the compiled sequence. While we could compile some special cases more efficiently (e.g., a sequence of consecutive integers like $\langle start..end \rangle$ can be compiled to a simple for-loop without any memory allocation), the LLVM backend has loop fusion, dead code elimination, and other optimisation passes that make this unnecessary. Since we do not have support for sets as first-class objects in our prototype, we cannot use this approach, and instead support special cases separately. In particular, we support ranges of the form $\{start..end\}$ and datatype names. Ranges are compiled to simple for-loops in the obvious way. Datatype names represent the set of values valid for the corresponding datatype, and the translation can require multiple for-loops. Specifically, we generate the code for each constructor in sequence, and use nested loops to represent the fields within each constructor (note that since these fields themselves can have a non-trivial type, they may need to similarly use multiple sequenced or nested loops); this corresponds directly to the notion of sum and product types in algebraic datatype theory.

```
1  // foreach v in e do { Body }, where e is a sequence
2  i ← 0
3  while i < length(e) do:
4      v ← e[i]
5      Body
6      i ← i + 1
7
8  // foreach v in {a..b} do { Body }
9  v ← a
10 while v ⩽ b do:
11     Body
12     v ← v + 1
13
14 // datatype X = C_0.A_0.A_1.....A_{n_a} | C_1.B_0.B_1.....B_{n_b} | ... | C_{m-1}.Z_0.Z_1.....Z_{n_z}
15 // foreach v in X do { Body }
16 foreach a_0 in A_0 do: // Translated recursively.
17     foreach a_1 in A_1 do:
18         ...
19         foreach a_{n_a} in A_{n_a} do:
20             v ← C_0.a_0.a_1.....a_{n_a}
21             Body
22 ...
23 foreach z_0 in Z_0 do:
24     foreach z_1 in Z_1 do:
25         ...
26         foreach z_{n_z} in Z_{n_z} do:
27             v ← C_{m-1}.z_0.z_1.....z_{n_z}
28             Body
```

Figure 3.12: Translations of loops over CSP_M types.

3.5.4 Sequences

The need to serialise and deserialise parameters, which may be sequences, at each step of a CSP$_M$ process necessitates a strict semantics for them. This allows us to represented sequences by a length coupled with an array of elements, as discussed in Section 3.4.2. In turn, this representation means that we can efficiently implement imperative-style `nth(index, sequence)` $\rightarrow$ `value` and `modify_nth(index, newval, sequence)` $\rightarrow$ `sequence` functions in addition to the traditional functional `length`, `head`, `tail`, and `concat` built-ins. All five of these functions are relatively straightforward to implement: `length` can directly read the length from the working representation; `head` and `nth` can extract the corresponding element; and `tail`, `modify_nth`, and `concat` can construct a new sequence, copying the necessary elements with the appropriate modification.

Sequences can be constructed directly with *list literals*, expressions of the form $\langle e_1, e_2, ..., e_n \rangle$, by compiling each of the component expressions and packing the results into a correspondingly sized array, with the length n fixed at compile-time. Alternatively, they can be constructed using a list comprehension of the form $\langle e_1, e_2, ..., e_n \mid s_1, s_2, ..., s_m \rangle$. List comprehensions are compiled using the loop techniques described in Section 3.5.3. Each set of bindings produced by the statements $s_1, ..., s_m$ is consumed by incrementing the sequence length by n and appending $e_1, ..., e_n$ (which typically have free variables bound by the statements) to the underlying array. Our prototype performs two iterations: one to compute the sequence length and allocate an appropriately sized array, and a second to actually populate the array. In some cases, the length can be computed without iterating over the statements. In practice, however, we have found that the LLVM optimiser can eliminate the loops in many cases, and we have not found it to be a significant overhead in other cases.

3.5.5 Functions

CSP_M supports generic functions parameterised by a number of potentially constrained type variables. LLVM, however, does not support generics.[7] We are unable to erase the generic argument types and use the same code for each instance since the different representations we use for different types can, for example, have incompatible copy semantics (e.g., Booleans can be copied directly, but sequences would need to allocate a new buffer, copy the elements, and then construct a new wrapper).

This leaves us to generate at least partially specialised code[8], but leaves some options of when to perform the specialisation. In particular, two obvious options are to compute the set of required specialisations at compile time, emitting the code for each as a function and specialising each call site, or to emit thunks in place of each call and generate necessary specialisations at run time. The former can require considerable overhead at compile time in the worst case, while the latter removes some opportunity for cross-procedural optimisation. Our implementation strikes a middle ground between the two by inlining all function calls; this allows us to avoid computing the specialisation sets explicitly while still enabling optimisation. However, this approach means that recursion is not allowed. To invoke such an inlined function, we compute the value of each of its arguments and bind it to the corresponding parameter patterns as described in Section 3.5.2, evaluate the body of the function in line, and pop the bindings.

3.6 Computing the Afters

While the previous sections focused primarily on non-process objects, this section deals with the processes themselves. We will first give a brief overview of the harness provided by FDR3, which is described in more detail in [GRABR15], and then proceed to

[7] At least not for user-defined functions; intrinsic functions such as the `llvm.memset.*` series can be *overloaded*, but each such specialisation set has to be built into the LLVM runtime, which is not suitable for our use case

[8] The .NET Common Language Runtime solves a similar problem by generating a specialisation for each value type and a single specialisation for all reference types.

```
function ProcessNode(n)
    for each (e, n′) ← afters(n) do
        HandleTransition(e, n′)
        if ShouldFollow(e, n′) then
            ProcessNode(n′)
                ▷ The node is typically queued up rather than processed immediately.
        end if
    end for
end function
```

Figure 3.13: High-level overview of FDR3's harness.

discussing the bulk of the lazy compiler, namely its VisitTransitions function.

At a very high level, FDR explores a GLTS one transition at a time, by querying each node's outgoing transitions and exploring some subset of them recursively (using a breadth-first search or a depth-first search), as shown in Figure 3.13. For example, it could only explore a strict subset of the transitions if the GLTS is combined with another GLTS that restricts the available events; however, this is outside the scope of this chapter. This effectively imparts a single-step semantics to the CSP_M process in question, but unlike executable programming languages, multiple possibilities might need to be returned for each step.

We model these semantics using a hybrid of *continuation-passing style* and *trampolines* on a *parameterised state machine*. The use of continuation-passing style naturally supports the multiple returns, since the continuation can simply be called on each returned value. Trampolines allow the harness to control execution step by step. The use of a parameterised state machine allows for an efficient implementation in which the trampolines are states rather than closures with executable code; additionally, it allows trampolines to be freely copied and even passed across the network for refinement-checking on a cluster. This is illustrated in Figure 3.14. Each node contains a tag identifying a *format* of the state machine and the *parameters* relevant to that format. Intuitively, the format represents a syntactic process expression like $a.x \longrightarrow P(x+1)$ and the parameters are the free variables in that expression, namely x in this example (assuming the a and P are a globally defined channel and process function, and

```
function ProcessNode(context, event, after)
    HandleTransition(event, after)
    if ShouldFollow(event, after) then
        ProcessNode(after, context,ProcessNode)
    end if
end function

function VisitTransitions(node, context, visitor)
    switch node.format do
        case [some state]
            Use node.[params] to compute afters(node)
            visitor(context, event_0, {format_0, params_0})
            visitor(context, event_1, {format_1, params_1})
            ...
        case [other state]
            ...
            visitor(context, ...)
            ...
    end switch
end function
```

Figure 3.14: High-level overview of the VisitTransitions function output by the lazy compiler, along with how it can be used in a system like FDR.

do not need to be recorded as mutable state). The rest of this section will explain these terms in more detail. VisitTransitions uses these to compute the *afters* of the corresponding node and returns the event-after pairs to its caller by calling the passed continuation, *visitor*. The caller handles the transition and if necessary passes the after back to VisitTransitions.

3.6.1 Computing relevant formats

While it might be tempting to use user-defined *named processes*, i.e. process expressions bound to a pattern or a function, as the formats, the single-step semantics mean this is not always possible. For example, consider the process $P = a \longrightarrow a \longrightarrow P$. Its sole after after a is $a \longrightarrow P$, which is not itself bound to a name. This example suggests the use of syntactic process expressions for unnamed processes. And indeed, we can take the idea further and use syntactic process expressions even for named processes,

```
global FA                                  ▷ Maps a format to the set of formats it can reach.
function ReachableFormats(P)
    switch P do                                                    ▷ Unwrap named processes
        case Q(a, ...), where Q(x, ...) = expr
            return ReachableFormats(expr)
        case Q, where Q = expr
            return ReachableFormats(expr)
        case else
            continue
    end switch
    if P ∈ FA then
        return FA(P)
    end if
    switch P do
        case Q □ R
            FA(P) ← ReachableFormats(Q) ∪ ReachableFormats(R)
        case □_i Q
            FA(P) ← ReachableFormats(Q)
        case b & Q
            FA(P) ← ReachableFormats(Q)
        case e ⟶ Q
            FA(P) ← Q ∪ ReachableFormats(Q)
        case else
            error
    end switch
    return FA(P)
end function
```

Figure 3.15: Recursively computes the syntactic formats a process can reach after one or more actions.

representing any named process by the expression it binds. This allows syntactically identical processes with different names to be identified with each other or identical unnamed processes.

In Figure 3.15 we present an algorithm for computing the formats reachable from a certain process expression, not including that expression itself. ReachableFormats first ensures that the argument is not a named process by unwrapping it as necessary. At this point, we are not concerned with function arguments since they are *parameters* rather than *formats*, so we treat process expressions bound to functions the same way

as those bound to patterns. Next, we check whether the requested process's reachable formats have already been computed and cached in order to avoid an infinite recursion. With these checks complete, we can handle actual process expressions. An external choice $Q \,\Box\, R$ can reach the formats either Q or R can reach, but not Q or R themselves, and likewise, a replicated external choice $\Box_i\, Q$ can reach the formats Q can reach but not Q itself. Similarly, a guarded expression $b \;\&\; Q$ can reach only those formats reachable from Q but not Q itself. From a prefix $e \longrightarrow Q$, however, we can reach both Q and its reachable formats. Having computed the appropriate selection of formats, we cache and return it. The lazy compiler can invoke ReachableFormats with the process it is compiling, and insert the process itself into the result to obtain the set of formats $\mathscr{F} = \textsc{ReachableFormats}(P) \cup \{P\}$ it needs to generate. The number of bits required for the format tag can be computed as $\lceil \log_2 |\mathscr{F}| \rceil$.

3.6.2 Compiling formats

The main task of the VisitTransitions function is to dispatch control to the appropriate format. This is straightforward; as in Figure 3.14, it's simply a matter of deserialising the format from the node's representation and switching based on its value. We will now discuss the implementation of the individual formats.

Prologue. At the start of each format's code, the parameters for that format are deserialised from the node's representation (which is packed), and each is stored in an LLVM register using the working representation described in Section 3.4. The code generator creates an environment mapping parameter names to the LLVM registers containing them and stores it in a stack; it is necessary to use chained environments since although the process expressions have single-step semantics, non-process expressions allow recursion and consequently a call stack. It is removed after the format is compiled.

External choice. From the operational semantics of external choice (given, e.g., in [Ros10]), we know that $\mathit{afters}(P \,\Box\, Q) = \mathit{afters}(P) \cup \mathit{afters}(Q)$. We can therefore generate the code for $P \,\Box\, Q$ as the concatenation of the code for P and Q.

Replicated external choice. The operational semantics of $\Box_i\, Q$, where i is a placeholder for a sequence of set statements, are such that $\mathit{afters}(\Box_i\, Q) = \bigcup_i \mathit{afters}(Q)$, and we can therefore generate the code for $\Box_i\, Q$ by compiling the statements as described in Section 3.5.3 with the bound values consumed by the code for Q.

Guarded expression. By definition, $b \;\&\; P = P$ if b is true and $b \;\&\; P = \mathit{STOP}$ otherwise. So, $\mathit{afters}(b \;\&\; P) = \mathit{afters}(P)$ if b and $\mathit{afters}(b \;\&\; P) = \emptyset$ otherwise. We generate code to compute the value of the expression b into a register and generate the code for P, guarded by a conditional on b.

Simple prefix. Prefix is the only presently supported operator that can actually introduce transitions; external choice and guarded expressions delegate that task to their component processes until eventually a prefix is reached. For a simple prefix (i.e., one without any input), we have $\mathit{afters}(e \longrightarrow P) = \{(e, P)\}$. To implement this, we generate code to serialise e to a buffer *event*, generate code to serialise P to another buffer *after* (this is distinct from generating code *for* P as in a format definition, since that would output P's afters rather than P itself), and emit code to call $\mathit{visitor}(\mathit{context}, \mathit{event}, \mathit{after})$. The event is serialised as described in Section 3.4, generating code to compute the value of each of its fields along the way.

To serialise a process, we must first determine its format and parameters. To enable this, if it is a named process, we *unwrap* it until reaching an unnamed syntactic process expression as shown in Figure 3.16. We can look up the unwrapped process expression P in $\mathscr{F}$ to find and serialise its format identifier. We then compute the free variables in P and serialise the value of each using the environment stack.

```
global FA                          ▷ Maps a format to the set of formats it can reach.
function UNWRAP(P)
    switch P do
        case Q(a, ...), where Q(x, ...) = expr
            Create a new environment with x ↦ a, ...
            return UNWRAP(expr)
        case Q, where Q = expr
            return UNWRAP(expr)
        case else
            return P
    end switch
end function
```

Figure 3.16: Recursively unwraps a named process to a syntactic expression.

Prefix with input. In a more general form of prefix, the left-hand side consists of an expression and a number of input or output *fields*, such as $e?x!y?z$, instead of a simple event expression. The fields are processed from left to right with each input field offering an *external choice* of the values matching its pattern and binding the values as appropriate. In other words, for example, $\mathit{afters}(e?x!y?z \longrightarrow P(x,z)) = \bigcup_x \bigcup_z \mathit{afters}(e.x.y.z \longrightarrow P(x,z))$.

More precisely, the left-hand side takes the form $e\langle f_1\rangle\langle f_2\rangle\langle ...\rangle\langle f_n\rangle$ where each f_i is of the form $?p$ for *unrestricted input* fields, $?p : S$ for *restricted input* fields, or $!v$ for output fields. Let $\mathscr{T}(f_i)$ be the set of allowable values for field f_i derived from the field's type. Then, we have

$$\mathit{afters}(e(?p)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P) = \bigcup_{v \in \mathscr{T}(?p)} \mathit{afters}(((e.v)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P)[p \mapsto v])$$

$$\mathit{afters}(e(?p : S)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P) = \bigcup_{v \in S} \mathit{afters}(((e.v)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P)[p \mapsto v])$$

$$\mathit{afters}(e(!v)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P) = \mathit{afters}((e.v)\langle f_1\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P)$$

with the base case for the recursion being the simple prefix already covered above. Code to evaluate these unions is implemented essentially as described in Section 3.5.3.

Nested loops are used to implement the unions corresponding to each input field. However, each field is serialised to the event buffer immediately, rather than at the

$\triangleright e\langle f_1\rangle\langle f_2\rangle\langle ...\rangle\langle f_n\rangle \longrightarrow P$
$event \leftarrow newbuffer$
$event \leftarrow serialise(event, e)$
for $v_1 \in \mathscr{R}(f_1)$ **do** $\triangleright$ where $\mathscr{R}(?p) = \mathscr{T}(?p)$; $\mathscr{R}(?p : S) = S$; $\mathscr{R}(!v) = \{v\}$
    If f_1 is an input field, register binding $p_1 \mapsto v_1$
    $event \leftarrow serialise(event, v_1)$
    ...
    for $v_n \in \mathscr{R}(f_n)$ **do**
        If f_n is an input field, register binding $p_n \mapsto v_n$
        $event \leftarrow serialise(event, v_n)$
            $\triangleright$ Below, P is serialised as for simple prefix, possibly using the bindings:
        $visitor(context, event, P)$
        If f_n is an input field, pop binding
    end for
    ...
    If f_1 is an input field, pop binding
end for

Figure 3.17: Translation of prefix with input.

innermost level; this results in significantly fewer redundant writes. This is shown in Figure 3.17, which uses *serialise* as a placeholder for the appropriate serialisation function defined in Section 3.4.1.

3.7 Performance

We will now proceed compare the performance of the lazy compiler presented in this chapter to the existing strict one. Since the lazy compiler is intended to be used only where the performance of the existing compiler is inadequate rather than as a default choice, most of the following benchmarks have been selected to highlight its strengths. In particular, they tend to use *parameterised state machines* resulting in a large number of possible instantiations only a small subset of which is actually reachable.

3.7.1 Needham-Schroeder Key Protocol

In this section, we consider the Needham-Schroeder Key Protocol (NSKP) using the CSP model presented in [Ros98]. We omit a description of the protocol itself and

instead focus on the modelling of a fully general attacker on the protocol with and without lazy compilation.

The attacker model used is the fully general model introduced in [RG97]. In this model, an attacker is parameterised by a set of *facts* that it knows. Further, it can *learn* a *message* by overhearing or intercepting it and it can *say* any of the messages it knows, where the messages make up a communicable subset of the facts. Further, it can use a protocol-specific set of *deductions* to learn facts it has not overheard directly. Each deduction is an inference rule of the form (X, f), meaning the fact f can be deduced from the set of facts X. We can define a function that computes the transitive closure of a set of facts with respect to a pre-defined set of deductions:

```
Close(S) = let S' = {f | (X,f) <- Deductions, diff(X,S)=={}}
           within if diff(S',S)=={} then S else Close(union(S,S'))
```

A spy can then be defined as `Spy(Known)` where `Known` is the set of prior knowledge (including, for example, public keys) and `Spy` is defined:

```
Spy(X) = learn?x -> Spy(Close(union(X,{x})))
      [] say?x:X -> Spy(X)
```

Here, `X` is a set of facts, while `learn` and `say` are channels of messages.

In terms of the strict evaluator traditionally used by FDR, this definition is intractable: the resulting state space is exponential in the number of facts, even though most of them are never reached in the complete system. In this NSKP model, there are 132 facts, and the state space is impossibly large. The typical workaround is to decompose the definition into a parallel composition with one process for each fact. An inference then requires a synchronised communication between the processes corresponding to its premises. A simplified version of the one given in [Ros98] is as follows:

```
ignorantof(f) = member(f,messages) & learn.f -> knows(f)
             [] infer?t:{(X,f') | (X,f') <- Deductions, f'==f}
                    -> knows(f)
```

```
knows(f) = member(f,messages) & say.f -> knows(f)
         [] member(f,messages) & learn.f -> knows(f)
         [] infer?t:{(X,f') | (X,f') <- Deductions, member(f,X)}
                 -> knows(f)

initial(f) = if member(f, Known) then knows(f) else ignorantof(f)

ParallelSpy' = (|| f:Facts @ [AlphaL(f)] initial(f)) \ {|infer|}
```

where `AlphaL` gives the alphabet of the corresponding process. A further optimisation can be made by separating the *learnable* facts from those known initially. Since the latter cannot be unlearned, they do not need to maintain any state or actively participate in inferences. This gives us the `Spy` defined in [Ros98], and which we will call `ParallelSpy` here.

In our testing, `ParallelSpy` is quite efficient, taking only 1-2 seconds to evaluate and find an attack on the protocol on the author's laptop. The strictly evaluated `Spy(Known)`, as expected, did not complete evaluation before exhausting memory.

Lazy-Compiled Spy. Due to the limited subset of CSP$_M$ supported by our prototype, particularly the incomplete support for sets, the author used a generator to emit part of the CSP$_M$ specific to this protocol. However, the generator does not use any problem-specific insight and in the future a similar translation could be implemented in FDR directly.

Since the lazy compiler is best suited to processes with integer or Boolean parameters, we expect a future implementation of sets to use a *bit vector*, with one bit per possible member to indicate its presence. For sets of small datatypes, this would have very efficient operations[9]. This does not work for sets of unbounded types such as

[9]For example, a set of 6-bit integers, ranging from 0 to 63, would require only 64 bits of storage regardless of how many are present. Computing the intersection of two such sets on a modern 64-bit CPU would only take a single, highly efficient `AND` instruction.

sequences, but for a universe of 132 possible facts, it's quite effective. As a result, our lazy spy is parameterised by a sequence of `card(Facts)` Booleans, rather than by a set of `Facts`. We expect support for mapping from CSP_M sets to an internal Boolean sequence representation to be added to FDR in the future, but for this example, we used a code generator to perform this translation externally.

Our prototype's limited support for functions, especially the lack of recursion, made implementing a `Close` function as written above impossible, and instead we implemented a step-by-step version using a hidden `infer` event. Additionally, to work around the lack of set support, we generated a hard-coded sequence of operations to implement the inferences.

The resulting spy is as follows:

```
LazySpy'(X) = ([] i : {29..118} @ learn'.i -> Spy(modify_nth(i,true,X)))
          [] ([] i : {29..118} @ nth(i,X) & say'.i -> Spy(X))
          [] infer -> Spy(<nth(0,X) or nth(2,X) or nth(3,X) or nth(4,X)
                          or nth(11,X) or nth(17,X) or nth(23,X), ...>)
```

Each fact is represented by an index into the sequence `X`. The range `{29..118}` represents the `messages` subset of the facts. The first line allows the spy to overhear or intercept any message and adds that message to the set `X`; note that the transitive closure of `X` is not computed yet. The second line allows the spy to communicate any of the messages it knows (`nth(i,X)` indicates whether the $\mathtt{i}^{\mathrm{th}}$ fact is known, and the `{29..118}` again restricts it to messages). The final line performs one step of inferences according to `Deductions`. We only include the deductions for the 0^{th} fact here for brevity. The `say'` and `learn'` channels are renamed and `infer` is hidden before placing the spy in the network:

```
Known' = <true, true, false, ...>
LazySpy =
    (lazy_compile(LazySpy'(Known'),{|say',learn',infer|})\{infer})[[
        say'.29 <- say.PK.pkA.Sq.Na.Alice,
```

```
        learn'.29 <- learn.PK.pkA.Sq.Na.Alice,
        say'.30 <- say.PK.pkA.Sq.Na.Bob,
        learn'.30 <- learn.PK.pkA.Sq.Na.Bob,
        ...
    ]]
```

Conclusion. We found the performance of our lazy compiler on this example to be quite promising. The combined run time of the code generator, the lazy compiler, and the refinement check was 1-2 seconds on the author's laptop. Given such a short run time, it's difficult to analyse in any detail, but the fact that it's as quick as the `ParallelSpy` suggests the lazy compiler to be quite suitable to similar applications.

We note that the problem studied in this section was intentionally small to present another alternative to the single-process `Spy`, which did not complete successfully due to its strictly explored untractable state space, rather than to compare `ParallelSpy` and `LazySpy` to each other. Indeed, with FDR's implementation of high-level machines, `ParallelSpy` has a similar representation to `LazySpy`, with the state of each fact recorded directly in the state vector of their parallel composition and the resulting state space explored lazily. We therefore do not contend that `LazySpy` offers any advantage over `ParallelSpy`, but we point out that `ParallelSpy` is an ingenious and problem-specific solution, whereas `LazySpy` can be generated automatically from `Spy`. Such decompositions may not be known or exist at all for other problems. The fact that our lazy compiler can handle a problem with 132 bits of state in seconds is promising for these problems.

We recognise that the presented `LazySpy` model is not quite as direct as the `Spy` model and would like to emphasise that adding the necessary support for sets is a straightforward though time-consuming exercise in programming. By using a generator, we demonstrate that a translation from sets to bit vectors is feasible and is only lacking integration with FDR.

3.7.2 Peg solitaire.

We now consider a larger example to evaluate the speed of refinement checking when using a lazy-compiled machine. As our baseline, we use a model of peg solitaire. Notionally, the model relies on a process representing the solitaire board with an event for each move and a `done` event when the solution is reached. However, owing to the size of the board, this results in more states than FDR's strict evaluator can handle. To work around this, the model described in [Ros10] decomposes the board into one process for each slot. The resulting system for the canonical board has over 187 million states and makes for a good benchmark. Using a depth-first search visits far fewer states and completes the check nearly instantaneously, so we will instead use a breadth-first search which does visit all the reachable states.

For the lazy-compiled version, we use the obvious single-process model below:

```
Solitaire(b) = b == TargetBoard & done -> Solitaire(b) []
    [] i:{0..Height-1}, j:{0..Width-1}, at(b,i,j)==P @ (
           valid(i+1,j) and valid(i+2,j) and
               at(b,i+1,j)==P and at(b,i+2,j)==E &
           up.i+2.j -> Solitaire(set_at(set_at(
               set_at(b,i,j,E),i+1,j,E),i+2,j,P))
       [] valid(i-1,j) and valid(i-2,j) and
               at(b,i-1,j)==P and at(b,i-2,j)==E &
           down.i-2.j -> Solitaire(set_at(set_at(
               set_at(b,i,j,E),i-1,j,E),i-2,j,P))
       [] valid(i,j+1) and valid(i,j+2) and
               at(b,i,j+1)==P and at(b,i,j+2)==E &
           right.i.j+2 -> Solitaire(set_at(set_at(
               set_at(b,i,j,E),i,j+1,E),i,j+2,P))
       [] valid(i,j-1) and valid(i,j-2) and
               at(b,i,j-1)==P and at(b,i,j-2)==E &
           left.i.j-2 -> Solitaire(set_at(set_at(
```

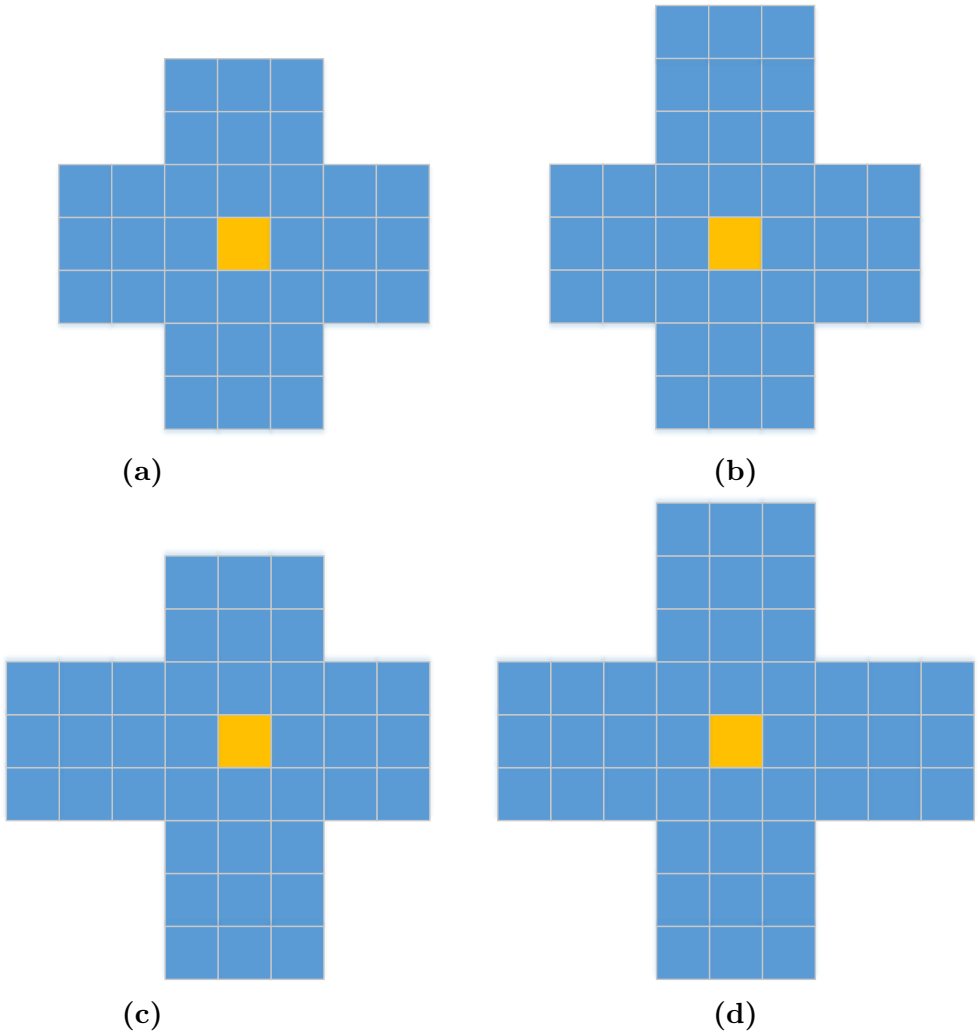


Figure 3.18: The peg solitaire boards used in our testing.

```
            set_at(b,i,j,E),i,j-1,E),i,j-2,P))
    )
```

In this representation, `b` is a sequence of slots, where each is either `E` meaning empty, `P` meaning full, or `X` meaning off-board. The `valid` function checks that the coordinates are valid, and the `at` and `set_at` functions access the slot at given coordinates. The first line checks if we have found the solution already. The next line iterates over each peg on the board and remaining lines offer to move that peg up, down, right, or left, respectively.

This lazy implementation has the same number of reachable states as the parallel

Table 3.1: Parameters and results for each peg solitaire board. A — indicates that the corresponding test was not run.

	States	Transitions	States/s (Lazy)	States/s (Parallel)	Memory (Lazy)	Memory (Parallel)
(a)	187M	1.487B	490K	3.01M	21.8 GB	3.5 GB
(b)	1.564B	13.971B	296K	2.13M	174 GB	17.5 GB
(c)	>808M	>10.056B	152K	—	250 GB	—
(d)	>536M	>7.918B	107K	—	250 GB	—

solution. Both solutions have a negligible compilation time, so we can compare the exploration speed directly. We tried a number of boards as shown in Section 3.7.2. Only the first two boards, (a) and (b), were soluble with the lazy compiler given 256 GiB of memory. The parallel versions were able to cover more of the state space, making a quantitative comparison unfair since the exploration speed varies throughout the check. We show the statistics for all four boards in Table 3.1.

Conclusion. We noted a significant increase in memory usage. This is due to the use of *lazy enumeration* which uses hash tables to map the structured states and events used by the lazy compiler to unique integers. In actuality, this is only necessary to support a class of problem in which the structure of the states is variable; for problems such as this one or state machines, the possible structures can be computed in advance and the more efficient representation used by FDR for supercombinator machines can be exploited. Preliminary testing suggests that this can in many cases improve speed and memory usage significantly, and can in some cases bring the memory usage to below that of a manually parallelised strictly evaluated version.

We observed a slowdown of less than one order of magnitude due to the use of lazy compilation in our prototype. This is sufficiently small that it should not be prohibitive, and it is also important to note that the comparison is between a mature and well-optimised implementation in the parallel case and a prototype with a number of obvious inefficiencies in the lazy case, which would be responsible for some of the performance loss. We discuss some of these inefficiencies in Section 3.8.

For comparison, when using the existing evaluator and the single-process version of

Board (a), evaluation did not complete in 48 hours. This illustrates the main strength of the lazy compiler: it brings the ability to present more natural models.

3.7.3 Double lock gate

As an illustration of what we might expect from a state machine compiler such as those used in industrial verification of embedded systems, we present a model of a double lock gate in a canal or a river. We will not discuss the system in great detail, but will illustrate its structure.

As a parameterised state machine, the model consists of a number of process functions taking integer or Boolean arguments and offering an external choice of simple prefix expressions going to one of the other states with some parameters. There are seven states similar to:

```
ClosedA(Lin,LoutA,LoutB,VOpen)=
    not VOpen& tock -> ClosedA(Lin,LoutA,LoutB,VOpen)
  []not VOpen& lockA -> Locked(Lin,LoutA,LoutB,VOpen)
  []not VOpen& openVA -> ClosedA(Lin,LoutA,LoutB,true)
  []VOpen& closeVA -> ClosedA(Lin,LoutA,LoutB,false)
  []VOpen& tock -> VStepA(Lin,LoutA,LoutB,VOpen)
  [](Lin==LoutA and not VOpen)& openGA -> OpenA(Lin,LoutA,LoutB,VOpen)
  []up_outA -> ClosedA(Lin,min(LoutA_h_limit,LoutA+1),LoutB,VOpen)
  []up_outB -> ClosedA(Lin,LoutA,min(LoutB_h_limit,LoutB+1),VOpen)
  []down_outA -> ClosedA(Lin,max(LoutA_l_limit,LoutA-1),LoutB,VOpen)
  []down_outB -> CClosedA(Lin,LoutA,max(LoutB_l_limit,LoutB-1),VOpen)
```

For our testing, we placed the system in parallel with a regulator that ensures the water levels (encoded in the state parameters) stay within certain levels. In Table 3.2, we compare the evaluation of the system as written with the use of the lazy compiler (Lazy), as written with the use of the strict evaluator (Strict), and using an involved decomposition into a number of parallel processes (Parallel). The resulting systems

Table 3.2: Memory usage, compilation time, and exploration time for each of the approaches to modelling the double lock system.

	Memory	Compilation (s)	Exploration (s)	Total (s)
Lazy	210 MB	1.2	1.5	2.7
Strict	3.4 GB	241	0.7	241.7
Parallel	213 MB	1.8	25.2	27

cover 2.02M states and 11.45M transitions.

Conclusion. We found that the lazy-compiled approach had both the lowest memory usage and the shortest overall run time. The exploration of the strictly compiled version of the same model was faster, but this came at the cost of a 200 times slower compilation. We therefore conclude that our approach has great potential for improving the model checking of industrial state machines.

3.7.4 Simple processor.

We now present a model of a very simple processor with N registers capable storing values from 0 to $M-1$. The processor receives an instruction on a channel and executes it. In our model, the available instructions are `Load.dst.val` to load a constant `val` into register `dst` or `Add.dst.src1.src2` to place the modulo-M sum of the contents of registers `src1` and `src2` into a register `dst`. Additionally, a `Read.r` instruction causes the processor to output the contents of register `r`.

We declare datatypes for registers and values to allow the lazy compiler to correctly deduce the number of bits required to store them:

```
datatype val = V.{0..M-1}
datatype reg = R.{0..N-1}
```

The *op* datatype declares the instructions supported by the processor:

```
datatype op = Read.reg
            | Load.reg.val
            | Add.reg.reg.reg
```

The processor reads instructions from the `exec` channel and potentially outputs values to the `out` channel:

```
channel exec : op
channel out : val
```

The processor itself is parametrised by a list of N register values of type `val` and implemented as follows:

```
CPU(regs) = exec?Read.r -> out!getr(r, regs) -> CPU(regs)
          [] exec?Load.dst.val -> CPU(setr(dst, val, regs))
          [] exec?Add.dst.src1.src2 -> CPU(setr(dst,
                 add_op(getr(src1, regs), getr(src2, regs)), regs))


getr(R.r, rs) = nth(r, rs) -- read register in file
setr(R.r, v, rs) = modify_nth(r, v, rs) -- write value to register
add_op(V.v1, V.v2) = V.((v1+v2) % M) -- implementation of Add op
```

Initially, all registers are 0:

```
StrictCPU = CPU(<V.0 | _ <- <0..N-1>>)
LazyCPU = lazy_compile(StrictCPU, {|exec, out|})
```

Now we create a program (a process that outputs a sequence of `exec` events) to output an infinite sequence of Fibonacci numbers modulo M starting with 0 and 1. Our implementation uses registers `R.0`, `R.1`, and `R.2` in rotation:

```
FibProgram = exec.Load.R.0.V.0 -> exec.Read.R.0
          -> exec.Load.R.1.V.1 -> exec.Read.R.1
          -> FibProgram'(0)
FibProgram'(n) = exec.Add.R.((n+2)%3).R.(n%3).R.((n+1)%3)
              -> exec.Read.R.((n+2)%3) -> FibProgram'((n+1)%3)
LazyFibProgram = lazy_compile(FibProgram,{|exec|})
```

To execute these programs on the processor, we simply put them in synchronised parallel:

```
StrictFib = (StrictCPU [|{|exec|}|] FibProgram) \ {|exec|}
LazyFib = (LazyCPU [|{|exec|}|] LazyFibProgram) \ {|exec|}
```

An example property one might wish to verify is that the CPU indeed outputs the Fibonacci sequence when executing this program. We can do so with the following equivalence checks:

```
-- Specification
Fib = Fib'(0,1)
Fib'(n1,n2) = out.V.n1 -> Fib'(n2, (n1+n2)%M)


assert Fib [T= StrictFib
assert StrictFib [T= Fib
assert Fib [T= LazyFib
assert LazyFib [T= Fib
```

Since the Fibonacci sequence modulo any integer is periodic, these are all finite-state processes.

We see in Figure 3.19 that the memory usage of the strict compiler grows with M^N while the lazy compiler uses roughly 150 MB regardless of the problem size. Figure 3.20 shows a similar pattern for compilation time. We have thus shown that the lazy compiler can take the problem from taking hours and using tens of gigabytes of memory to well under a second and not much more memory than FDR3 uses when idle.

Experimenting further, we found that the lazy compiler can compile the problem with $N = 5$ and an astounding $M = 30000$ in 24 seconds using 1135 MB of memory. This is largely due to the alphabet size (which grows as $N^3 + NM + N + M$), however, and simply printing out the event set takes a similar amount of time and memory.

Since our refinement check is effectively linear (consisting of a sequence of hidden `exec.Add`, hidden `exec.Read`, and visible `out` events), the refinement checking speed

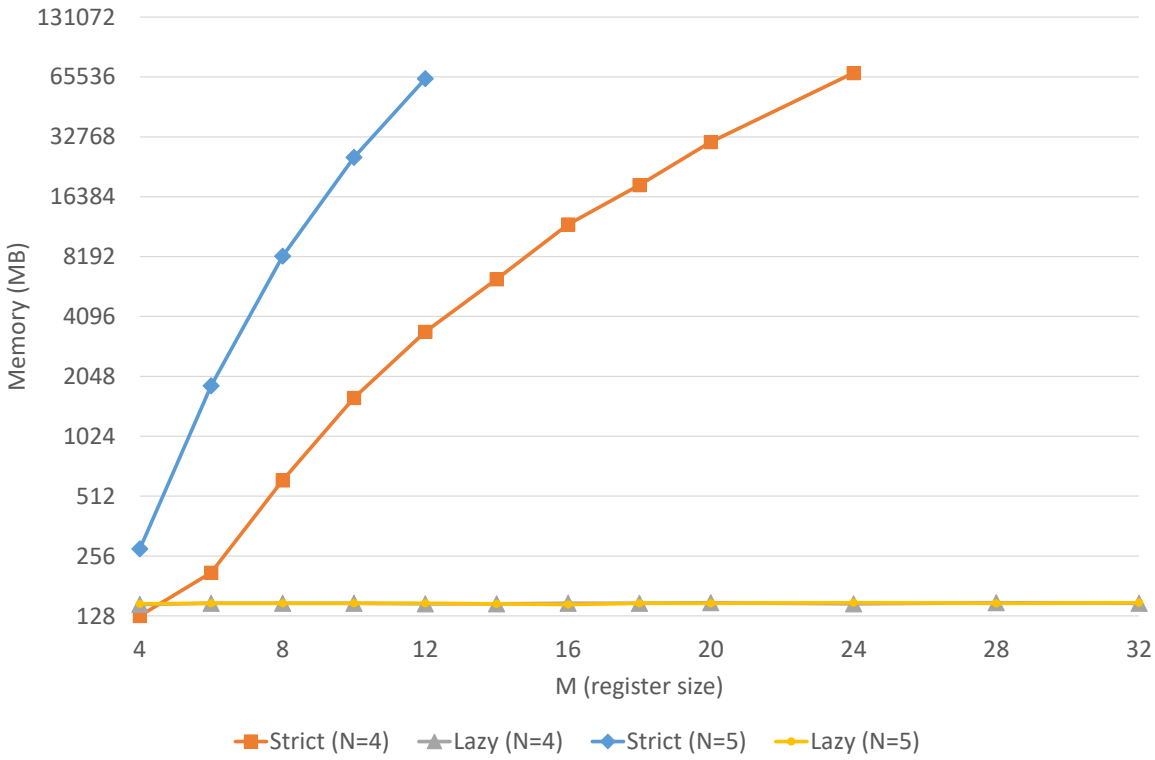


Figure 3.19: Memory used by the strict and lazy compilers on the processor model running the Fibonacci program with a range of parameters.

is limited by the time it takes to set up and tear down each ply of the breadth-first search. Moreover, the number of states checked is limited to 3 times the period of the resulting Fibonacci sequence, which in turn is at most $6M$.[10] We therefore do not include timings for the refinement checking phase of this example.

Conclusion. We have shown that the lazy compiler is able to handle a class of problems that FDR's existing compiler struggles with both for time and memory. We observed that for the larger lazy-compiled examples, the compilation time is dominated by the upfront compilation of the structured event alphabet into FDR's internal flat representation. This suggests that performance could further be improved by adding support for structured events to FDR's refinement checker. This would additionally remove the need to pass an alphabet to the `lazy_compile` operator and moreover allow

[10]Showing this is an interesting exercise in number theory, and a proof can be found, e.g., at `http://www.mathpages.com/home/kmath078/kmath078.htm`. For the purposes of this thesis, all that is relevant is that the number of explored states was no larger than $18M$ plus a small constant.

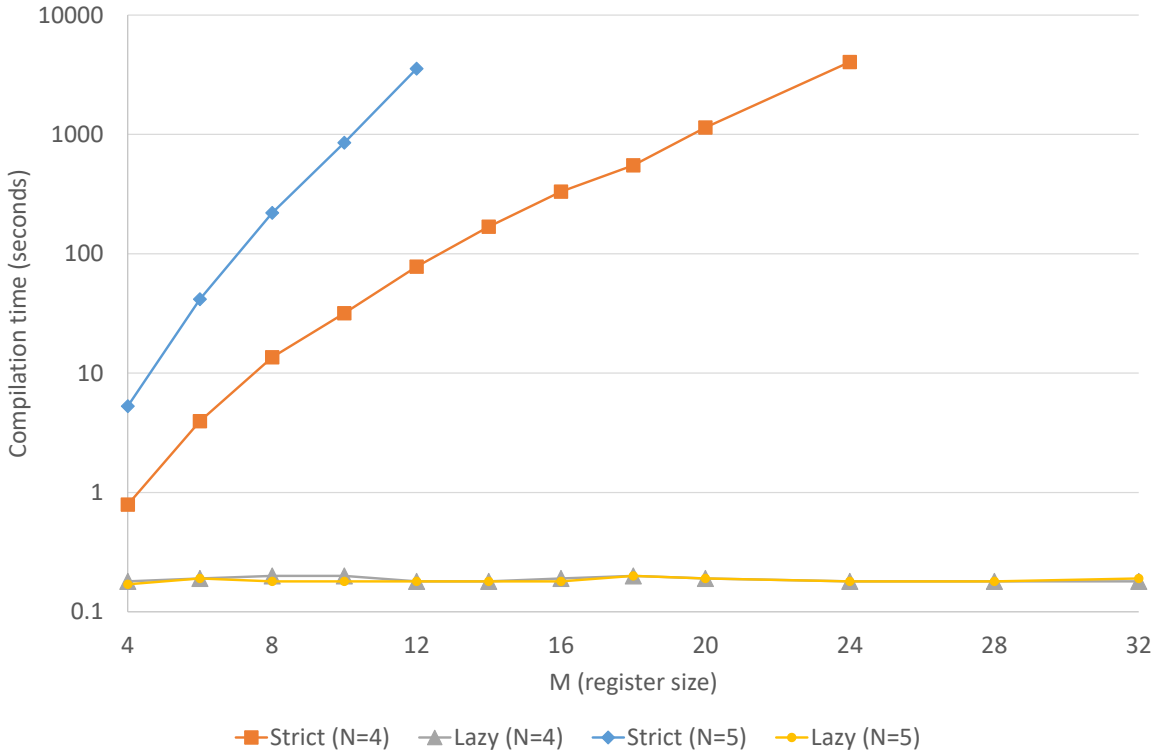


Figure 3.20: Time taken by the strict and lazy compilers on the processor model running the Fibonacci program with a range of parameters.

the use of statically unbounded alphabets, such as those an infinite counter process might use.

3.8 Future Work

We are aware of a number of potential enhancements for our lazy compiler as well as some inefficiencies in the prototype that can be remedied. We discuss these here.

3.8.1 Performance and scalability improvements

Lazy enumeration. At present, much of the performance cost comes from the *lazy enumeration* used to bridge the lazy-compiled states to the rest of FDR. The refinement checker presently requires a list of fixed-length *formats* for states, and lazy enumeration allows us to translate an arbitrary-length lazy-compiled state into a 32-bit integer.

This has a number of performance and scalability implications. Most obviously, it

increases the memory usage considerably, as it requires storing each state three times: the lazy-compiled key in the hash table, the lazy-enumerated value in the hash table, and the lazy-enumerated state again in the refinement checker's list of seen states. Moreover, depending on the parameters of the hash table used, there is either a space or a speed cost. There is an additional cost in terms of parallelisation: although using a *sharded* hash table allows us to benefit from FDR's multi-threaded refinement checking, it does not support its cluster mode. Besides the space usage, the hash table insertions and lookups both take time, which grows with the size and the number of states.

In terms of scalability, FDR's existing lazy enumeration facility outputs 32-bit state identifiers, which limits individual lazy-enumerated machines to 4 billion states. Traditionally, such large component machines have not been common due to FDR handling parallel compositions of small machines much more efficiently, but we have shown in Section 3.7.2 that with lazy compilation, machines coming close to the limit can be practical. This hard limit can of course be extended by emitting 64-bit state identifiers instead. However, there are still practical limits on the number of states due to the requirement for a hash table to be held in RAM, which limits the space used by the sum of all lazy compiled machines. One of FDR's strengths is its ability to efficiently explore systems with large numbers of states by using the disk, so this is an important concern.

This lazy enumeration is only needed for systems where the possible state formats are not known until run time. For example, the buffer process:

```
BUFF(xs) = #xs > 0 & out!head(xs) -> BUFF(tail(xs))
        [] in?x -> BUFF(<x>^xs)
```

can require storage of an arbitrary number of xs. Moreover, a compact representation of an algebraic data type such as:

```
datatype List = Empty | Full.<Int>
```

might use only 1 bit for an empty list but an arbitrary amount of space for full lists; representing the sequence `<List>` would then require an unknown amount of space

even if its length is known. However, for many practical examples, neither of these are the case. The intruder and solitaire examples in Section 3.7.1 and Section 3.7.2 both use sequences with a statically known number of statically-sized elements. We could therefore avoid lazy enumeration in these and similar cases using fairly simple static analysis. In order to support cases where the formats are indeed arbitrary, we could investigate adding support for arbitrary-length states directly into the refinement checker.

Data representation. We use a simple bit-streaming approach to represent the data used in events and states, as detailed in Section 3.4. This can waste a large amount of space: given a Boolean followed by two integers, 31 bits of padding will be inserted between the two integers, a nearly 50% overhead. A similar situation can arise when dealing with sequences of datatypes. The datatype:

```
datatype Either = Left.Int | Right.Int
```

would be stored with 1 bit to indicate the constructor followed by a 32-bit integer for either constructor's field. A sequence of `Either` elements would then use an average of 33 bits of each 64-bit word.

A different packing algorithm can reduce this overhead significantly. However, if the algorithm is used at run time, it can affect performance negatively. It is worth investigating whether there are packing algorithms such that their efficiency would justify their cost, and whether we can pre-compute more efficient packings at compilation time given statically determinable formats as above; the *supercompiler* does something similar for supercombinator machines. It is also worth investigating whether it would be more efficient to dispense with 64-bit alignment and avoid any padding.

Dynamic events. Though CSP_M events are structured, FDR relies on them being converted to 32-bit integers before use and moreover relies on the alphabet of a process being fully available before the process is used.

The alphabet is needed upfront primarily for the efficient implementation of supercombinators. It is worth investigating whether the requirement could be removed in other cases such as top-level or enumerated processes. Since the lazy compiler cannot know in advance what events it might emit at runtime, we currently request the alphabet from the user. This is not ideal for usability, but more importantly results in a tradeoff between safety and performance in the case where the alphabet is not correct: since the lazy compiler cannot in general detect violations statically (any potential static violation could be mitigated at runtime by putting the process in parallel with $STOP$), we must either trust the user or check that each transition's event is allowed. Moreover, if an efficient way to support structured events in supercombinators is discovered, that would expand the range of use cases considerably.

Adding support for structured events directly to FDR is orthogonal to this in that we could construct an alphabet with structured events or compile events to the unstructured form lazily without computing an alphabet upfront. Structured events would free the lazy compiler from keeping track of an event map and performing the necessary translation for each transition. It would also leave room for other symbolic machine types; for example, it could yield a more efficient way to rename entire channels as it often necessary in protocol models.

Emitted code. The use of LLVM allows for a wide range of optimisations. For our prototype, we used a default sequence of optimisation passes tuned for C++ compilers. It is likely that this is not the most well-suited for our output. Moreover, it is possible that we could use CSP-specific knowledge to emit optimisation hints where some information might have been lost during compilation.

Strict compilation. Our lazy compiler is primarily aimed at cases where the speed is limited by the unvisited states that would need to be visited by FDR's strict evaluator. However, we have observed that even when there are few such states, the lazy compiler can perform better than the strict evaluator due to interpretation and other overhead. It would be interesting to investigate whether a strict compiler could offer an even bigger

performance advantage for such cases. We do, however, expect that lazy compilation followed by enumeration should not be much slower.

3.8.2 Extensions

Sets. It would be useful to add support for sets over small domains implemented as bit sets. We have shown that this can be both effective and useful in Section 3.7.1 using a manual translation.

Recursive non-process functions. As mentioned in Section 3.5.5, our compiler implements non-process functions by inlining, and consequently does not support recursion. We would like to investigate whether it is possible to remove this restriction efficiently.

User-defined operators. It would be interesting to investigate whether user-defined CSP-like operators (as in [Ros11, Ros15, Ros10]) could be efficiently supported by this lazy compiler. It is difficult to support them in FDR's present evaluation model due the use of optimised representations for each operator. However, a compiler could emit similarly optimised representations for operators defined using arbitrary sets of combinators. This would also allow support for other CSP operators such as internal choice to be added more easily.

3.9 Conclusions

We have presented an efficient lazy compiler for CSP_M using a dynamic translation to LLVM. We produced a prototype implementation of this compiler fully integrated into FDR and allowing it to be used in conjunction with the existing strict evaluator even within a single model.

We demonstrated that it allows users to present more natural models for a wide range of systems with large parameter spaces. We have evaluated the compilation and refinement checking performance on typical systems and found it to be sufficient to

justify its use where it allows more natural formulations of a problem. We found that in some cases characteristic of industrial examples of state machines, lazy compilation results in a speed and memory improvement both over strictly evaluating the same model and over a carefully crafted parallel decomposition designed to play to FDR's strengths.

Chapter 4

Bisimulation

4.1 Introduction

Many different variations on bisimulation have been described in the literature on process algebra, for example [Par81, Mil81, vGW96, PU96, San96]. They are typically used to define equivalences between nodes of a labelled transition system (LTS), but they can also be used to calculate state-reduced LTSs that represent equivalent processes[1]. FDR typically views a large process as the parallel composition of a number of component processes, which are often sequential. The resulting LTS is contained within the Cartesian product of the components' LTSs. One of the approaches it takes to combat the state explosion problem is to supply a number of compression functions that attempt to reduce the state spaces of these components.

The set of compressions described in [RGG+95], which included *strong* bisimulation, has been extended by several other versions of bisimulation in the most recent versions of FDR. In this chapter, we introduce novel algorithms for computing maximal *divergence-respecting delay bisimulations* (DRDB) and *divergence-respecting weak*

[1]Necessarily, any technique for reducing the number of states will lose some information about the inputs, potentially making presentation of counterexamples in terms of the input non-trivial. As discussed in [RGG+95], FDR's debugger gets around this with the help of additional refinement checks: to investigate the behaviour of a compressed process in a certain counterexample, FDR performs a refinement check of the uncompressed process against a specification that permits all behaviours except for the counterexample.

bisimulations (DRWB) based on dynamic programming and incorporating the idea of *change tracking* introduced in the context of strong bisimulation in [BO05]. These algorithms are the first efficient algorithms for directly computing the maximal weak and delay bisimulation relations, without constructing an expensive intermediate form. For obvious reasons the choice of algorithms for FDR is primarily influenced by practical efficiency (provided the algorithms are correct, of course), and we include comprehensive benchmark results. These show that our new algorithms are much more efficient than existing algorithms used by other tools on many examples. We also compare the performance of a number of strong bisimulation algorithms.

Weak bisimulation is a well-known relation in which chains of invisible τ actions are elided into a single τ, and where chains of τs before and after a visible action a are absorbed into a. Delay bisimulation is the same as weak bisimulation except that only τs before visible a are absorbed. Thus delay bisimulation is a finer relation than weak bisimulation. We can make both these relations finer yet by insisting that they do not identify a divergent state with a non-divergent one: this is what is meant by *divergence-respecting*. We consider the divergence-respecting variants to make the resulting compressions consistent with CSP semantics; our algorithms work equally well for the non-divergence-respecting versions of these bisimulations.

All of the discussed bisimulation relations preserve CSP semantics, as well as the operational semantics of programs in other languages with operational semantics described by such GLTSs and relying only on observational equivalence. They can therefore be used to combat the space explosion problem faced in explicit model checking for such languages.

Model checkers frequently use branching bisimulation [vGW96] due to the existence of an efficient $O(nt)$ algorithm [GV90] and the absence, prior to our own work reported here, of a sufficiently efficient algorithm to compute the even coarser weak bisimulation directly. Even though DRDB and DRWB compress better than branching bisimulation they typically give less compression than FDR's pre-existing compressions. However

the latter are not sound for some functionality of FDR[2]. In Section 4.4.5 we compare these two classes of compressions. The new algorithms are highly effective in practice. *Change tracking* can significantly reduce wasted effort, and the use of dynamic programming to compute *afters* on the fly for the divergence-respecting delay and weak bisimulations typically gives a vast reduction on memory usage and time for transition systems with many τs: this can be as much as several orders of magnitude.

Section 4.2 first defines terms used throughout the rest of this chapter. The rest of the section summarises *iterative refinement*, which is discussed in its naïve form in [BO02] and in an optimised form using *change tracking* in [BO05], and the Paige-Tarjan algorithm [PT87, Fer90] as implemented by FDR. Since our DRDB and DRWB algorithms build on these foundations, the reader is recommended to study this section before reading the subsequent sections. Divergence-respecting delay and weak bisimulations are closely related, and we present them together in Section 4.3. Finally, we compare the time and compression characteristics of each of our algorithms and a number of alternatives on various benchmarks in Section 4.4.

4.2 Background

As explained in Section 2.2, FDR uses LTSs in which nodes sometimes have additional behaviours represented by labellings such as divergences or minimal acceptances. The algorithms presented in this chapter input and output GLTSs, and the definitions of the various bisimulations are modified to only identify states with identical node labels. However, regular LTSs are supported as a special case. As input, an LTS $(N, \Sigma, \longrightarrow)$ is equivalent to the GLTS $(N, \Sigma, \longrightarrow, \{\emptyset\}, \lambda)$, where $\lambda(n) = \emptyset$ for all $n \in N$. The output GLTSs have node labellings corresponding to those of the input, transformed to account for the different domains, and possibly divergence markings. If the input is an LTS, rather than a GLTS, and an output LTS is to be created then τ self-loops could be used in place of divergence markings.

[2]Use inside the `prioritise` operator and with semantic models richer than failures. This includes virtually all Timed CSP examples and most other real-time CSP models.

4.2.1 Strong Bisimulation

Definition 4.2.1. Given a GLTS $G = (N, \Sigma, \longrightarrow, \Lambda, \lambda)$, a relation $R \subseteq N \times N$ is a *strong bisimulation* of G if and only if it satisfies all of the following:

$$\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_1 \xrightarrow{x} m_1$$
$$\Rightarrow \quad \exists m_2 \in N \cdot n_2 \xrightarrow{x} m_2 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2, m_2 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_2 \xrightarrow{x} m_2$$
$$\Rightarrow \quad \exists m_1 \in N \cdot n_1 \xrightarrow{x} m_1 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2 \in N \cdot n_1\ R\ n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$

Two nodes are *strongly bisimilar* if and only if there exists a strong bisimulation that relates them. The *maximal strong bisimulation* on a GLTS S is the relation that relates two nodes if and only if they are strongly bisimilar. The FDR function `sbisim` computes the maximal strong bisimulation on its input GLTS and returns a GLTS with a single node bisimilar to all of the nodes in each equivalence class in the input. FDR has included the `sbisim` compression function since the release of FDR2. Other bisimulation relations are defined similarly, differing in their use of a derived transition relation in place of $\longrightarrow$ and potentially additional constraints on related nodes beyond equivalence of their node labels.

4.2.2 Naïve Iterative Refinement

The FDR2 implementation of `sbisim` first computes the desired equivalence relation as a two-directional one-to-many map between equivalence class and node identifiers. (This map allows us to quickly determine the index of any node's equivalence class, and the set of nodes denoted by any such index.) It then generates a new GLTS based on the input and the computed equivalence relation. This final step is straightforward to implement and dependent more on the internal GLTS format than the bisimulation algorithms and so will not be discussed here in much detail. Furthermore, it is not specific to strong bisimulation and can be used to factor a GLTS by an arbitrary

```
1: function IterativeRefinement(Approximation, ComputeAfters, Refine)(G)
2:     ρ ← Approximation(G)
3:     repeat
4:         ρ′ ← ρ
5:         afters ← ComputeAfters(G, ρ′)
6:         ρ ← Refine(G, ρ′, afters)
7:     until ρ = ρ′
8:     ConstructMachine(G, ρ)
9: end function
```

Figure 4.1: The iterative refinement skeleton used by most of our bisimulation algorithms. We present it as a curried higher-order function so that implementers can pass the functions specifying a certain algorithm and users would pass the GLTS it is to be run against.

equivalence relation. Computing the desired equivalence relation is the more interesting problem and the topic of this chapter.

We will use the term *coloured afters* to refer to the *afters* of a node m under the last iteration's equivalence relation ρ, that is, $\{(e, \rho(n)) \mid m \xrightarrow{e} n\}$. This is similar to the *signatures* introduced by Blom and Orzan in [BO02]. Wimmer et al. present a list of *signatures* for a number of variants of bisimulation in [WHH+06]; we supplement them with node labels to support GLTSs and divergence-sensitive models.

A very high level overview of iterative refinement is illustrated in Figure 4.1. A coarse approximation of the equivalence relation is first computed by *Approximation*, usually using the first-step behaviour of each node, and each class in this relation is repeatedly refined using the first-step behaviours of the nodes under the current approximation. This is related to the formulation of strong bisimulation given in [Mil81] as a series of experiments of increasing depth and is the naïve method mentioned by Kanellakis and Smolka in [KS83].

1: **function** NullApproximation($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
2: **return** $\{(n, 0) \mid n \in N\}$ ▷ Put all the nodes in one class.
3: **end function**

(a) NullApproximation puts all nodes into the same equivalence class.

1: **function** FirstStepApproximation($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
2: $\Phi \leftarrow \langle (n, \mathit{initials}(n), \lambda(n)) \mid n \in N \rangle$
3: Sort Φ by $(n, a, l) \mapsto (a, l)$
▷ We use this notation to mean sorting a sequence of (n, a, l)s by (a, l).
▷ After the sort, nodes with equivalent *initials* and node labels are adjacent.
4: $a', l' \leftarrow \mathit{INVALID}, \mathit{INVALID}$
5: $c \leftarrow 0$
6: **for each** $(n, a, l) \leftarrow \Phi$ **do**
7: **if** $a \neq a' \vee l \neq l'$ **then**
▷ Different *initials* or node label; start a new partition.
8: $a', l' \leftarrow a, l$
9: $c \leftarrow c + 1$
10: **end if**
11: $\rho \leftarrow \rho \cup \{(n, c)\}$
12: **end for**
13: **return** ρ ▷ ρ partitions N into the following equivalence classes:
▷ $\{\{n' \mid (n', a', l') \in \Phi \wedge a' = a \wedge l' = l\} \mid a \in \Sigma, l \in \Lambda \cdot \exists n \cdot (n, a, l) \in \Phi\}$
14: **end function**

(b) The more involved FirstStepApproximation instead classifies nodes based on their first-step behaviour: their node labels and initial events.

Figure 4.2: Two initial approximation functions, each returning a partition function $\rho : N \to \mathbb{N}^+$.

4.2.2.1 Algorithm

Initial approximation. The initial approximation can most simply be computed by identifying all nodes, as does NullApproximation in Figure 4.2a. Instead, FDR3 uses the finer FirstStepApproximation in Figure 4.2b to compute an initial approximation $\rho_0 = \{\{n \in N \mid \lambda(n) = \lambda(m) \wedge \mathit{initials}(n) = \mathit{initials}(m)\} \mid m \in N\}$ based on the nodes' labels and *initials*. This is equivalent to identifying all nodes and then performing one refinement using the nodes' labels and their *coloured afters*. Unlike the *afters* of a node, whose colour and therefore equivalence depends on the current equivalence relation, these node labels are fixed and we can save time by only comparing them once. The time savings can be significant depending on the size of

```
1: function ComputeAllAfters((N, Σ, ⟶, Λ, λ), ρ)
2:     cafters ← ⟨(n, {(a, ∅) | ∃m · n -a-> m}) | n ∈ N⟩
3:     for each n -a-> m do
4:         cafters(n)(a) ← cafters(n)(a) ∪ {ρ(m)}
5:     end for
6:     return cafters
7: end function

8: function RefineAll((N, Σ, ⟶, Λ, λ), ρ, cafters)
9:     Sort cafters by (n, A) ↦ A.
10:    (n′, A′) ← INVALID, INVALID
11:    c ← max(ρ)
                ▷ Instead of computing max(ρ), we could persist c across iterations.
12:    for each (n, A) ← cafters do
13:        if A ≠ A′ ∧ ρ(n) = ρ(n′) then
                        ▷ Different coloured afters in the same partition; start new one.
14:            c ← c + 1
15:        end if
16:        ρ(n) ← c
17:        n′, A′ ← n, A
18:    end for
19:    return ρ
20: end function

21: function SBisimNaïve := IterativeRefinement(
                FirstStepApproximation, ComputeAllAfters, RefineAll)
```

Figure 4.3: Naïve Iterative Refinement for strong bisimulation.

the node labels; in particular some GLTSs in FDR have nodes labelled with *minimal acceptances* drawn from the set $\Lambda = \mathcal{P}(\mathcal{P}(\Sigma))$. In addition to the time savings, a side effect of this finer initial approximation is that an algorithm that is not aware of node labellings, such as that given in [Fer90], can be used for the iteration phase.

Iteration. Assume that we have already separated the nodes into equivalence classes, whether from the initial approximation or from a previous refinement step. We will now attempt to refine these classes further. We first compute the *afters* of each node *coloured* per the latest equivalence relation, as shown in ComputeAllAfters in Figure 4.3. Then, we use these *coloured afters* to produce a more refined equivalence relation as illustrated in RefineAll. This sorts the *coloured afters* for the nodes in

```
1: function ConstructMachine((N, Σ, ⟶, Λ, λ), ρ)
2:     cafters ← ComputeAllAfters((N, Σ, ⟶, Λ, λ), ρ)
3:     N′ ← range(ρ)                          ▷ Create a node for each equivalence class.
4:     λ′ ← ∅
5:     T′ ← ∅
6:     for each n ∈ N′ do
7:         n′ = pick(ρ⁻¹(n))                  ▷ Compute a representative node for the class.
8:         λ′ ← λ′ ∪ (n, λ(n′))
9:         for each (a, M) ∈ cafters(n′) do
10:            for each m ∈ M do
11:                T′ ← T′ ∪ (n, a, m)                          ▷ Compute transitions.
12:            end for
13:        end for
14:    end for
15: end function
```

Figure 4.4: The ConstructMachine function for strong bisimulation. The *pick* function chooses an arbitrary member of its nonempty set argument. Note that instead of recomputing *cafters*, we could use the *cafters* already computed by RefineAll.

each class (line 9), and a single in-order traversal through the sorted lists (lines 10 to 18) then allows us to reclassify the nodes in each class.

If any nodes have changed class during this pass, we must proceed to refine the classes again. Otherwise, we are done. We can determine whether any nodes have changed class during the final reclassification traversal with very little additional work.

Construction. The final step is to construct the output GLTS as outlined in Figure 4.4. To do this, we first create a node for each equivalence class (line 3). Any node labels can be copied from an arbitrary representative in each class (the n' in line 7), as they are guaranteed to be equivalent by the initial approximation (line 8). Next, we output a transition corresponding to each input transition. This can generate duplicates, and we must take care to only output one copy of each. Instead of using the input transitions directly, we also have the option of using the already computed *coloured afters* to create the transitions (lines 9 to 13), using an arbitrary representative from each class since the *coloured afters* for each of the nodes in an equivalence class are guaranteed to be equivalent (since the refinement phase has terminated).

4.2.2.2 Representation of *coloured afters*

Blom and Orzan [BO02] state a worst-case complexity in $O(nt)$, where the input has n nodes and t transitions. They are able to achieve this by assuming a bounded fanout, which allows the representation of *coloured afters* to be ignored. Since our primary objective is developing algorithms that perform well on practical examples as typified by our extensive benchmarks on real machines, rather than seeking the best asymptotic behaviour on notional ones, we must choose a data structure that works well in practice.

Asymptotically a tree representation of *coloured afters* sets seems most efficient, with $O(\log c)$ time for each insertion, where c is the number of equivalence classes in the output GLTS. However, in practice, we have observed in nearly all cases a significant speedup from using sorted arrays instead, with $O(c)$ time for each insertion. This is likely due to the *coloured afters* sets often being much smaller than c, and due to the x86 architecture being optimised for operations on contiguous blocks of memory. For this reason, most of the performance results in Section 4.4 refer only to implementations using sorted arrays.

4.2.3 Change-Tracking Iterative Refinement

We will now present the strong bisimulation algorithm FDR3 uses, an improvement on Naïve Iterative Refinement. With some bookkeeping, we can determine which states' *coloured afters* could not possibly have changed after the previous iteration. The algorithm shown in Figures 4.5 to 4.7 uses this information to avoid recomputing and sorting the *coloured afters* for these states, in a similar fashion to the optimisation used by Blom and Orzan in [BO05]. Since FDR represents states as consecutive integers and the transitions are stored in an array, we can easily construct a constant-time accessible map from nodes to their predecessors. A node then might change class at the $n + 1^{th}$ iteration only when it is one step back from a node that has changed class on the n^{th}.

We will maintain the following items as running state: a bit vector *changed* containing the nodes whose equivalence class changed on the previous iteration and a bit

```
1: global changed
                ▷ Output from RefineChanged to ComputeChangedAfters.
2: global affected
                ▷ Output from ComputeChangedAfters to RefineChanged.
3: global cafters
    ▷ Read and written by ComputeChangedAfters; persisted for optimisation.
4: global T^{-1}   ▷ Read by ComputeChangedAfters; persisted for optimisation.

5: function SBisimCT((N, Σ, ⟶, Λ, λ))
6:     changed ← N
7:     affected ← ∅
8:     cafters ← ⟨(n, {(a, ∅) | ∃m · n -a-> m}) | n ∈ N⟩
9:     T^{-1} ← {(n, {m | ∃a · n -a-> m}) | n ∈ N}
10:    return IterativeRefinement(FirstStepApproximation,
                   ComputeChangedAfters, RefineChanged)
               ((N, Σ, ⟶, Λ, λ))
11: end function
```

Figure 4.5: Change-Tracking Iterative Refinement entry point for strong bisimulation. Note that this presents and initialises a number of global variables used by ComputeChangedAfters and RefineChanged.

vector *affected* containing the nodes that might be affected by those changes. Before the first iteration, *changed* should be initialised with all nodes marked since the initial approximation classified all the nodes. This initialisation is done by SBisimCT in Figure 4.5.

On each iteration we will start by recalculating the changed *coloured afters* as shown in Figure 4.6. First, we need to compute *affected* by iterating through *changed* and adding each of their predecessors (lines 3 to 6). We then clear (lines 8 to 10) and recompute (lines 11 to 13) the *coloured afters* for each of the nodes in *affected*. All nodes that are not marked for update get to keep their *coloured afters* from the previous iteration.

Next, we refine the equivalence classes as shown in Figure 4.7. We compute the equivalence classes that contained the affected nodes in the previous iteration (line 5); these are the equivalence classes that might need to be refined, and this can be computed in linear time in the number of nodes by iterating over *affected*. We must also clear *changed* for the next step.

```
1: global changed, affected, cafters, T^{-1}
2: function ComputeChangedAfters((N, Σ, ⟶, Λ, λ), ρ)
3:     affected ← ∅
4:     for each n ∈ changed do
5:         affected ← affected ∪ T^{-1}(n)
6:     end for

7:     for each n ∈ affected do
8:         for each (a, M) ∈ cafters(n) do
9:             cafters(n)(a) ← ∅
10:        end for
11:        for each (a, m) ∈ {n -a-> m} do
12:            cafters(n)(a) ← cafters(n)(a) ∪ {ρ(m)}
13:        end for
14:    end for
15:    return cafters
16: end function
```

Figure 4.6: ComputeChangedAfters used by Change-Tracking Iterative Refinement for strong bisimulation. See also Figure 4.5 for the global variables referenced here.

```
1: global changed, affected
2: function RefineChanged((N, Σ, ⟶, Λ, λ), ρ, cafters)
3:     changed ← ∅
4:     c ← max(ρ)                ▷ As in RefineAll, we could persist c across iterations.
5:     for each class ∈ {ρ(n) | n ∈ affected} do
6:         nodes ← ρ^{-1}(class)
7:         Reorder nodes by moving those not in affected to the front.
8:         Sort remaining nodes (those in affected) by n ↦ cafters(n).
9:         Find largest run with equivalent cafters and remove it from nodes.
10:        A' ← INVALID
11:        for each n ∈ nodes do
12:            if A ≠ A' then        ▷ Different coloured afters; start a new partition.
13:                A' ← A
14:                c ← c + 1
15:            end if
16:            ρ(n) ← c
17:            changed ← changed ∪ {n}
18:        end for
19:    end for
20:    return ρ
21: end function
```

Figure 4.7: RefineChanged used by Change-Tracking Iterative Refinement for strong bisimulation. See also Figure 4.5 for the global variables referenced here.

For each of the classes that we consider for refinement, we first separate the nodes that have not changed class from those that have, the latter being in *affected* (line 7). Next, we sort the nodes that are in *affected* and in this class in order to partition this class (line 8). Once we have sorted the *coloured afters* of all of the affected nodes in a given class, we choose the *largest* sequence of nodes with the same *coloured afters* to keep the original class index (line 9). We assign new indices to the rest (lines 10 to 18), rather than picking the *first* such sequence. We must also record the nodes that had new indices assigned in *changed* (line 17).

If *changed* is empty, we can conclude that we have reached a fixed point, and we can terminate the algorithm, returning the bisimulation relation we have computed implicitly in the equivalence class indices of the nodes.

4.2.4 Paige-Tarjan Algorithm

We have also implemented the algorithm outlined in [Fer90]. This is an adaptation of Paige and Tarjan's solution (described in Section 3 of [PT87]) to the relational coarsest partition problem (which is equivalent to single-action strong bisimulation) that works with LTSs by splitting with respect to each element of the alphabet in sequence whenever the original algorithm would split a class. In summary, each time a class is split, the resulting subclasses are recorded. Refinement is then performed with respect to the initial classes (separating nodes with edges into each class from those without) and with respect to each split class (separating nodes with edges into one subclass, the other, or both) using the inverse labelled transition relation.

We have produced two implementations of this algorithm for performance comparison. The first implementation stays true to the original formulation, which makes heavy use of linked lists. An alternative implementation uses arrays like our other algorithms.

4.2.4.1 Complexity

The worst-case time complexity for a GLTS with n nodes and t transitions is in $O(t \log(n))$. However, the cached in-counts (the *info maps* of [Fer90]) necessary to achieve this bound can be unwieldy to manipulate, raising the implementation and runtime costs. In addition, as the algorithm requires frequent construction and traversal of sets, there is a time or space penalty depending on the set representation used. In fact, in our performance tests (Section 4.4.2 and Table 4.2), change-tracking iterative refinement outperforms the Paige-Tarjan algorithm despite the latter's superior asymptotic time complexity. This is consistent with Blom and Orzan's observations in [BO05].

4.3 Divergence-Respecting Delay and Weak Bisimulations

While FDR has supported strong bisimulation since version 2, it has only added support for variants of weak bisimulation in version 2.94. This was because the weak bisimulation of [Mil81] is not compositional for most CSP models and because FDR already had compressions that successfully eliminated τ actions, the most notable of which are `normal` and `diamond`. `normal` was originally provided as an important component of refinement checking in FDR, but has since also proven to be a useful compression function in its own right. However, it does sometimes create an exponential increase in the size of a LTS. `diamond` [RGG+95] is a compression that was designed to remove redundant *diamonds* of transitions in LTSs, but is not semantics-preserving when combined with prioritisation, Timed CSP, and stronger semantic models such as refusal testing. Due to the limitations of both `diamond` and `normal`, FDR2.94 [AGL+12] implemented a new `dbisim` compression (originally called `wbisim`), which returns the maximal divergence-respecting delay bisimulation (DRDB) of its input.

Although FDR3 does not support refinement checking in models other than the

traces, (stable) failures, and failures-divergences models $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{N}$ directly, it can nevertheless be necessary to preserve semantics in richer models in intermediate results. The priority operator introduced in [RAH12] allows certain events in $\Sigma^{\tau\checkmark}$ to be given *priority* over others, such that in any state, an event is only allowed to be performed if no higher priority event is available. In general, this depends on the acceptance sets (or, in a more restricted variant of priority, refusal sets) in each node traversed on a given path, whereas the failures model $\mathcal{F}$ only records the refusal at the end of a given trace. Consequently, compressions designed for $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{N}$, such as `diamond`, can lose information necessary for the priority operator.

While the priority operator is interesting on its own, its introduction into FDR was largely motivated by Timed CSP [RR88]. The implementation of Timed CSP in FDR is detailed in [ALOR12] and relies on *digitisation* and a translation to the discretely-timed *tock*-CSP. The result of the translation is a plain CSP process with a distinguished *tock* event that happens regularly and abstracts the passing of time. In the absence of priority, *tock*-CSP can suffer from a lack of progress; namely, when a τ is available from an idling state (one where the process can perform a *tock* without changing state, allowing an arbitrary amount of time to pass), the process is free to repeat the *tock* forever. To resolve this, the principle of *maximal progress* central to *tock*-CSP dictates that if a τ and a *tock* are available from the same state, the τ is to be given priority, which, as the general case of the priority operator, requires at least the refusal testing model $\mathcal{RT}$. Applying a compression that is not compositional with $\mathcal{RT}$ before placing a process in a timed priority context can therefore produce incorrect results.

Strong bisimulation, of course, preserves the necessary information, but it does not take advantage of the fact that τ is not externally observable, and so is not very effective when dealing with systems that have a large number of τ transitions. This section will introduce *divergence-respecting delay bisimulation* and *divergence-respecting weak bisimulation*, which are both compositional with all CSP models and can provide more compression than strong bisimulation.

4.3.1 Definitions

Definition 4.3.1. Given the transition relation $\longrightarrow$ of a GLTS S, let us define a binary relation $\Longrightarrow$ such that $p \Longrightarrow q$ if and only if there is a sequence $p_0, ..., p_n$ (with n possibly 0) such that $p = p_0$, $q = p_n$, and $\forall\, i < n \cdot p_i \xrightarrow{\tau} p_{i+1}$. Let us further define a ternary relation $\hookrightarrow$ with $p \overset{a}{\hookrightarrow} q$ for $a \in \Sigma$ if and only if $\exists p' \cdot p \Longrightarrow p' \wedge p' \xrightarrow{a} q$, and $p \overset{\tau}{\hookrightarrow} q$ if and only if $p \Longrightarrow q$. This is the *delayed transition relation*, since the visible events are delayed by 0 or more τs.

Definition 4.3.2. A relation $R \subseteq N \times N$ is a *divergence-respecting delay bisimulation* of a GLTS S if and only if it satisfies all of the following requirements:

$$\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_1 \overset{x}{\hookrightarrow} m_1$$
$$\Rightarrow \quad \exists m_2 \in N \cdot n_2 \overset{x}{\hookrightarrow} m_2 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2, m_2 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_2 \overset{x}{\hookrightarrow} m_2$$
$$\Rightarrow \quad \exists m_1 \in N \cdot n_1 \overset{x}{\hookrightarrow} m_1 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2 \in N \cdot n_1\ R\ n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$

$$\forall\, n_1, n_2 \in N \cdot n_1\ R\ n_2 \quad \Rightarrow \quad n_1\Uparrow \Leftrightarrow n_2\Uparrow$$

Note that the definition is very similar to that of strong bisimulation. The differences are the use of the delayed transition relation and the added clause about divergence, which is necessary to make the compression compositional for CSP[3]. However, if we precompute divergence information and record it in each node's label, the requirement that $n_1 \Uparrow \Leftrightarrow n_2 \Uparrow$ will be absorbed into the requirement that $\lambda(n_1) = \lambda(n_2)$.

FDR3 adds support for compression by the even coarser divergence-respecting weak bisimulation (DRWB).

Definition 4.3.3. Given the transition relation $\longrightarrow$ of a GLTS S and the binary relation $\Longrightarrow \equiv \xrightarrow{\tau}{}^*$, let us define a ternary relation $\Longrightarrow$ with $p \overset{a}{\Longrightarrow} q$ for $a \in \Sigma$ if and

[3]Of all the semantic models of CSP, only the simple traces model $\mathcal{T}$ identifies an LTS node with no action and one whose only action is τ to itself. These two nodes are both weakly and delay bisimilar under the usual definitions.

only if $\exists p', q' \cdot p \Longrightarrow p' \wedge p' \xrightarrow{a} q' \wedge q' \Longrightarrow q$, and $p \overset{\tau}{\Longrightarrow} q$ if and only if $p \Longrightarrow q$. This is the *observed transition relation.*

Definition 4.3.4. A relation $R \subseteq N \times N$ is a *divergence-respecting weak bisimulation* of a GLTS S if and only if it satisfies all of the following requirements:

$$\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_1 \overset{x}{\Longrightarrow} m_1$$
$$\Rightarrow \quad \exists m_2 \in N \cdot n_2 \overset{x}{\Longrightarrow} m_2 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2, m_2 \in N \cdot \forall\, x \in \Sigma^\tau \cdot n_1\ R\ n_2 \wedge n_2 \overset{x}{\Longrightarrow} m_2$$
$$\Rightarrow \quad \exists m_1 \in N \cdot n_1 \overset{x}{\Longrightarrow} m_1 \wedge m_1\ R\ m_2$$

$$\forall\, n_1, n_2 \in N \cdot n_1\ R\ n_2 \quad \Rightarrow \quad \lambda(n_1) = \lambda(n_2)$$

$$\forall\, n_1, n_2 \in N \cdot n_1\ R\ n_2 \quad \Rightarrow \quad n_1{\Uparrow} \Leftrightarrow n_2{\Uparrow}$$

Note that the definition is very similar to that of divergence-respecting delay bisimulation. The only difference is the use of the observed transition relation in place of the delayed transition relation.

The FDR3 compression function `wbisim` computes the maximal DRWB on its input GLTS and returns a GLTS with a single node DRW-bisimilar to all of the nodes in each equivalence class in the input. It is an important compression because, like `sbisim` and `dbisim` it preserves semantics in all CSP models, while potentially offering a higher amount of compression than `dbisim`. Weak bisimulation is also important because (at least in its non divergence-respecting form) it and strong bisimulation are the best known and most studied bisimulations in the literature. This compression is new to FDR3 and is the strongest implemented compression for CSP models richer than the failures model and the other cases where `diamond` is not semantically valid.

4.3.2 Reduction to Strong Bisimulation for DRDB and DRWB

The definition of DRDB suggests a reduction to strong bisimulation. For an input GLTS S, we can compute a GLTS $\widehat{S}$ with a transition for each delayed transition of

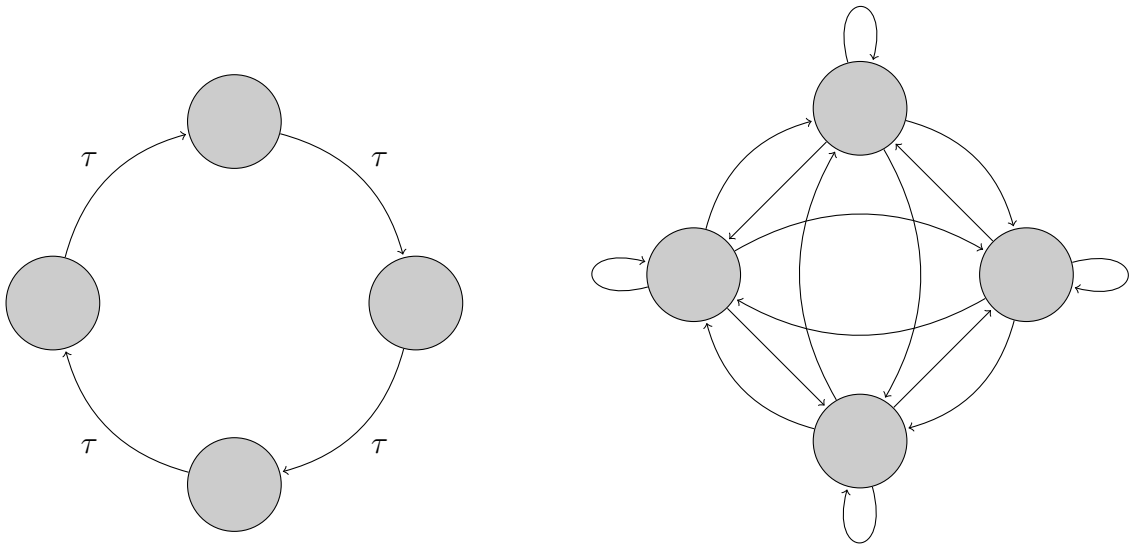
```
1: function DRDBisimulationReduction((N, Σ, ⟶, Λ, λ))
2:     λ′ = ∅
3:     for each n ∈ N do
4:         λ′(n) ← (λ(n), CheckDivergence(n))
5:     end for
6:     Compute ↪ from ⟶.
7:     return SBisimCT((N, Σ, ↪, Λ × {Divergent, NotDivergent}, λ′))
                                                  ▷ Or SBisimNaïve.
8: end function
```

Figure 4.8: An implementation of DRD-bisimulation by reduction to strong bisimulation. The DRW-bisimulation is similar, but uses $\Longrightarrow$ instead of $\hookrightarrow$.

the input and mark each node with divergence information computed from S. Care is required not to introduce divergences not present in S due to the τ self-loops introduced in $\widehat{S}$ because the original node can take an empty sequence of τs to itself. The maximal strong bisimulation of $\widehat{S}$ is the maximal DRDB (respectively, DRWB) of S by construction, and this can be computed by any of the algorithms described in Section 4.2.1, as shown in Figure 4.8. FDR2.94 employs such a reduction to compute a maximal DRDB, and uses naïve iterative refinement for the strong bisimulation step.

Complexity. A significant problem with this approach is the high worst-case space complexity. $\widehat{S}$ can have up to An^2 transitions if the input has n nodes and an alphabet of size A, even if S has as few as An transitions. For example, a process that performs N τs before recursing exhibits this worst-case behaviour. Since all nodes are mutually τ-reachable, a transition system with N^2 transitions is constructed. Figure 4.9 demonstrates this quadratic explosion for $N = 4$.

Construction of $\widehat{S}$ can take a correspondingly significant amount of time. For example, using an adaptation of the Floyd-Warshall algorithm [Flo62] requires $O(n^3)$ operations. The strong bisimulation step after this transformation will take a correspondingly large amount of time ($O(An^3 \log n)$, for naïve iterative refinement). Regardless of the strong bisimulation algorithm used, the memory usage is likely to be prohibitive.



(a) The input, $P(4)$, has only four transitions and four nodes.

(b) The output has sixteen transitions for the same four nodes. Labels have been omitted for clarity.

Figure 4.9: The constructed LTS can be quadratically larger than the input.

Divergence-Respecting Weak Bisimulation. It is possible to compute a maximal DRWB with a similar reduction, creating $\widehat{S}$ with the observed transitions of S instead of its delayed transitions. Given our results from testing with DRDB, we do not believe that such an implementation would offer any improvement over DRDB, and we have therefore not created one.

4.3.3 Dynamic Programming Approach for DRDB

Rather than constructing $\widehat{S}$ and keeping it in memory (which is often the limiting factor for such computations, since main memory is limited and the hard disk is prohibitively slow given the random nature of the accesses required by parts of the strong bisimulation algorithm), FDR3 instead recomputes the relevant information using the original transition system on each refinement iteration. To the knowledge of the authors of [BGRR16], this the first algorithm that avoids calculating $\widehat{S}$ explicitly.

1: **global** *toponodes*
▷ Read by ComputeAllDelayedAfters; persisted for optimisation.

2: **function** DBisimDP(G)
3: $(N, \Sigma, \longrightarrow, \Lambda, \lambda) \leftarrow$ TauLoopFactor(G)
4: *toponodes* $\leftarrow N$ sorted topologically according to $\xrightarrow{\tau}$
5: **return** IterativeRefinement(DBisimApproximation, ComputeAllDelayedAfters, RefineAll) $((N, \Sigma, \longrightarrow, \Lambda, \lambda))$
6: **end function**

7: **function** ComputeAllDelayedAfters($(N, \Sigma, \longrightarrow, \Lambda, \lambda), \rho$)
8: *cafters* $\leftarrow \langle (n, \{(a, \emptyset) \mid \exists m \cdot n \overset{a}{\hookrightarrow} m\}) \mid n \in N \rangle$
▷ Can be cached across iterations.
9: **for each** $n \leftarrow$ *toponodes* **do**
10: *cafters*$(n)(\tau) \leftarrow \{\rho(n)\}$ ▷ Implicit self-loop.
11: **for each** $(a, m) \in \{(a, m) \mid n \xrightarrow{a} m\}$ **do** ▷ Visible *afters*.
12: *cafters*$(n)(a) \leftarrow$ *cafters*$(n)(a) \cup \{\rho(m)\}$
13: **end for**
14: **for each** $m \in \{m \mid n \xrightarrow{\tau} m\}$ **do** ▷ Delayed *afters*.
15: **for each** $(a, M) \in$ *cafters*(m) **do**
16: *cafters*$(n)(a) \leftarrow$ *cafters*$(n)(a) \cup M$
17: **end for**
18: **end for**
19: **end for**
20: **return** *cafters*
21: **end function**

Figure 4.10: The dynamic programming algorithm for DRDB, with the straightforward DBisimApproximation omitted due to space constraints.

4.3.3.1 Algorithm

The algorithm is shown in Figure 4.10. First, noting that two nodes on a τ loop are both DRD-bisimilar and divergent, we factor the input GLTS S by the relation that identifies nodes on a τ loop[4] (line 3). FDR has a function built in that does this, `tau_loop_factor`. We will not discuss it in detail here, but it uses Tarjan's algorithm for finding strongly connected components [Tar72] via a single depth-first search and runs in $O(n + t)$ time for a system with n nodes and t transitions. In addition to eliminating τ loops, it marks each node as divergent or stable. Now that we have ensured there are no τ loops, the τ-transition relation can be used to topologically sort the nodes with another depth-first search [Tar76], so that there are no upstream τ-transitions (line 4).

The topological sort allows us to obtain the transitions of the $\widehat{S}$ described in Section 4.3.2 using a dynamic programming approach. The most downstream node in this topological sort has no outgoing τ transitions, so its new *initials* and *coloured afters* are precisely those in S (lines 11 to 13) with the addition of itself after τ (line 10). We then proceed upstream and for each node compute the union of its own *coloured afters* (with the inclusion of a self-transition under τ) and the *coloured afters* of each of the nodes it can reach under a single τ transition (lines 14 to 18). Of course, since we are doing this in a topological order, these nodes have been processed already, so we have computed the union of the *coloured afters* of all τ-reachable nodes from the given node.

We can apply a modified Naïve Iterative Refinement (Section 4.2.2) to compute the maximal strong bisimulation of $\widehat{S}$, which is itself never constructed (line 5). We compute the *initials* and node labels for the initial approximation using dynamic programming on the topologically sorted nodes. For each refinement, we compute the *coloured afters* using the dynamic programming approach described above. For the

[4]Even if the nodes on a τ loop have different different labels, their mutual reachability means that it is impossible to distinguish them in any of the CSP models, so the compression is sound as long as the resulting node's label captures all the possible behaviours (e.g., if the node labels are *minimal acceptances*, `tau_loop_factor` would output their minimised union).

construction step, we compute the equivalence classes of the *coloured afters* as above, but without inserting the τ self-transition.

4.3.3.2 Complexity

The space complexity for this algorithm is never significantly higher than that of the explicit reduction, and can be significantly lower. The only additional information we have is the transient DFS stack and bookkeeping information, and the sorted node list. The *coloured afters* we compute for each node, which are sets of equivalence class identifiers, take no more space than the exploded transition system, and will take less if any nodes are identified – and if the user is running the algorithm there is reason to believe that they will be. In addition, since the *coloured afters* are recomputed at each iteration, the working set for each refinement iteration can be smaller than the peak working set required by the final one. For example, for the process $P(N)$ portrayed in Figure 4.9, the initial classification will identify all nodes, and the first *coloured afters* computation will have a single *after* for each node: equivalence class 0 under τ.

We still traverse the entire transition set a single time (split across nodes). But now, for each node, we have to take the union of its *coloured afters* and the ones preceding it. Provided we keep these sorted, and use a merge sort for union, we will have in the worst case $O(Acn)$ operations for each node, where A is the size of the alphabet, c is the number of classes in this iteration, and n is the number of nodes, since Ac is the maximal number of *coloured afters* a node could have and we could have $O(n)$ nodes following this one. This means an upper bound on the overall worst-case runtime is $O(An^3c)$.

However, in practice the time complexity is much lower. Removing τ loops ensures that the graph is not fully connected and reduces the number of unions for each node significantly in systems with divergence, which eliminates many worst cases. The number of classes c is often much less than n. In addition, there are further optimisations that could be made to reduce the runtime, the union operation can be made faster by keeping metadata that allows us to avoid computing the unions of duplicate

```
1: global toponodes
          ▷ Read by ComputeAllObservedAfters; persisted for optimisation.

2: function ComputeAllObservedAfters((N, Σ, ⟶, Λ, λ), ρ)
3:     cafters ← ⟨(n, {(a, ∅) | ∃m · n ↪^a m}) | n ∈ N⟩
                                                  ▷ Can be cached across iterations.
4:     for each n ← toponodes do      ▷ Compute nodes reachable in 0 or more τs:
5:         cafters(n)(τ) ← {ρ(n)}
6:         for each m ∈ {m | n ⟶^τ m} do
7:             cafters(n)(τ) ← cafters(n)(τ) ∪ cafters(m)(τ)
8:         end for
9:     end for
10:         ▷ Compute nodes reachable in 0 or more τs, 1 visible event, 0 or more τs:
11:    for each n ← toponodes do
12:        for each (a, m) ∈ {(a, m) | n ⟶^a m ∧ a ≠ τ} do
13:            cafters(n)(a) ← cafters(n)(a) ∪ cafters(m)(τ)
14:        end for
15:        for each m ∈ {m | n ⟶^τ m} do
16:            for each (a, M) ∈ cafters(m), a ≠ τ do
17:                cafters(n)(a) ← cafters(n)(a) ∪ M
18:            end for
19:        end for
20:    end for
21:    return cafters
22: end function
```

Figure 4.11: The dynamic programming algorithm for DRWB, with the straightforward WBisimApproximation and entry-point WBisimDP omitted due to space constraints.

coloured afters sets (though we do not currently employ such optimisations). Section Section 4.4.3 demonstrates that the dynamic programming approach is faster on many examples with a large number of τs than the explicit reduction approach.

4.3.4 Dynamic Programming Approach for DRWB

We proceed in a manner similar to that described in Section 4.3.3. Noting that two nodes on a τ loop are both DRW-bisimilar and divergent, we factor the input GLTS by the relation that identifies nodes on a τ loop using `tau_loop_factor`. We then topologically sort the nodes by the τ-transition relation.

The topological sort allows us to obtain the observed transitions (recall Defini-

tion 4.3.3) using the two-pass dynamic programming approach in Figure 4.11. One pass, as in delay bisimulation, is not sufficient since we need to determine the *coloured* τ^* *afters* of the visible *afters* of each node, and these visible *coloured afters* might not have been previously explored. In the first pass, we compute the *coloured* τ^* *afters* of each node (lines 4 to 9). The last node in this topological sort has no outgoing τ transitions, so its only τ^* *after* is itself. We then proceed upstream and for each node compute the union of its own *coloured* τ *afters* (with the inclusion of its own equivalence class) and the previously computed *coloured* τ^* *afters* of each of the nodes it can reach under a single τ transition. The second pass computes the visible observed transitions. For each node, these are the union of the *coloured* τ^* *afters* of its visible *afters* (lines 12 to 14) and the visible observed transitions of its τ *afters* (lines 15 to 19). If we proceed in topological order, the visible observed transitions of each node's τ *afters* will have already been computed by the time they are needed.

We can apply a modified Naïve Iterative Refinement to compute the maximal strong bisimulation of the induced GLTS as described in Section Section 4.3.3, removing the τ self-transition from each node in the construction step.

4.3.4.1 Complexity

In the typical case this algorithm will require more space to store the *coloured afters* than the DRDB algorithm since it must follow the τ transitions after a visible event in addition to the ones tracked by the DRDB algorithm. However, the worst-case space complexity for this algorithm is the same, since in the worst case all the nodes are mutually reachable under both the delayed transition relation and the observed transition relation. The time complexity is a constant factor greater since at each iteration two passes through the topologically sorted nodes must be performed.

In practice we have found that `wbisim` is nearly as fast as `dbisim`, and produces identical results on nearly all inputs.

1: **global** *toponodes*
    ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.
2: **global** *changed*
    ▷ Output from RefineChanged to ComputeChangedDelayedAfters.
3: **global** *affected*
    ▷ Output from ComputeChangedDelayedAfters to RefineChanged.
4: **global** *cafters* ▷ Read and written by ComputeChangedDelayedAfters.
    ▷ Persisted for optimisation.
5: **global** T^{-1}
    ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.
6: **global** T_τ^{-1}
    ▷ Read by ComputeChangedDelayedAfters; persisted for optimisation.

7: **function** DBisimCTDP($(N, \Sigma, \longrightarrow, \Lambda, \lambda)$)
8: $(N, \Sigma, \longrightarrow, \Lambda, \lambda) \leftarrow$ TauLoopFactor(G)
9: *toponodes* $\leftarrow N$ sorted topologically according to $\xrightarrow{\tau}$
10: *changed* $\leftarrow N$
11: *affected* $\leftarrow \emptyset$
12: *cafters* $\leftarrow \langle (n, \{(a, \emptyset) \mid \exists m.n \overset{a}{\hookrightarrow} m\}) \mid n \in N \rangle$
13: $T^{-1} \leftarrow \{(n, \{m \mid \exists a \in \Sigma.n \xrightarrow{a} m\}) \mid n \in N\}$
14: $T_\tau^{-1} \leftarrow \{(n, \{m \mid n \xrightarrow{\tau} m\}) \mid n \in N\}$
15: **return** IterativeRefinement(DBisimApproximation,
ComputeChangedDelayedAfters, RefineChanged)
$((N, \Sigma, \longrightarrow, \Lambda, \lambda))$
16: **end function**

Figure 4.12: Entry point to the dynamic programming algorithm for DRDB with change tracking. Note that this presents and initialises a number of global variables used by ComputeChangedDelayedAfters and RefineChanged. We do not explicitly mention *affected_topo* for brevity, since it is always updated together with *affected*.

```
1: global toponodes, changed, affected, cafters, T^{-1}, T_τ^{-1}
2: function ComputeChangedDelayedAfters((N, Σ, ⟶, Λ, λ), ρ)
3:     affected ← ∅
4:     for each n ∈ changed do
5:         affected ← affected ∪ T^{-1}(n) ∪ {n}
6:     end for
7:     Add all nodes reachable from affected via T_τ^{-1} to affected using, e.g., a DFS.
8:
9:     for each n ∈ affected_topo do
10:        for each (a, M) ∈ cafters(n) do
11:            cafters(n)(a) ← ∅
12:        end for
13:        cafters(n)(τ) ← {ρ(n)}                          ▷ Implicit self-loop.
14:        for each (a, m) ∈ {(a, m) | n --a--> m} do      ▷ Visible afters.
15:            cafters(n)(a) ← cafters(n)(a) ∪ {ρ(m)}
16:        end for
17:        for each m ∈ {m | n --τ--> m} do                ▷ Delayed afters.
18:            for each (a, M) ∈ cafters(m) do
19:                cafters(n)(a) ← cafters(n)(a) ∪ M
20:            end for
21:        end for
22:    end for
23:    return cafters
24: end function
```

Figure 4.13: The dynamic programming algorithm for DRDB with change tracking. See also Figure 4.12 for the global variables referenced here

4.3.5 Change-Tracking with Dynamic Programming for DRDB and DRWB

FDR3.2 improves this dynamic programming approach to computing *coloured afters* with an adaptation of the change tracking introduced in [BO05] and discussed here in Section 4.2.3.

4.3.5.1 Two-Pass Change-Tracking DRDB

The idea behind the two-pass approach is to separate the change-tracking and the dynamic programming aspects into two separate passes at each iteration. Figure 4.12 shows how the existing iterative refinement skeleton can be reused with a new ComputeChangedDelayedAfters function illustrated in Figure 4.13.

As is necessary for the dynamic programming component, we factor the input by the relation that identifies nodes on a τ loop using `tau_loop_factor` and topologically sort the nodes by the τ-transition relation, recording a bidirectional map from nodes to their indices in the sort. As in change-tracking iterative refinement for strong bisimulation, we will maintain *changed* and *affected* bit vectors, the former initialised to contain all the nodes in the input. So that we can efficiently traverse *affected* in topological order, we will also maintain an *affected_topo* bit vector containing the indices of affected nodes under the topological sort. We do not want to create a map from nodes to their predecessors under the delayed transition relation due to space considerations: such a map could be as large as the transition set of the derived GLTS $\widehat{S}$, which could be significantly larger than the input GLTS. Instead, we will create a map from nodes to their predecessors under a single visible event and another map from nodes to their predecessors under a single τ. We will then use these two maps to compute the affected nodes.

Initial Approximation. The initial approximation can be computed by dynamic programming as in Section 4.3.3. To start, we record that the label for the most downstream node in the topological sort according to $\widehat{S}$ is as per S and its *initials* are

as per S with the addition of τ. We then proceed upstream and for each node record its *initials* under $\widehat{S}$ as the union of $\{\tau\}$, its *initials* in S, and the *initials* of each of its τ-successors in $\widehat{S}$ (which have already been computed owing to the topological order of our computation); its label is computed similarly.

Iteration. As outlined in Figure 4.13, we first need to determine which nodes need their *coloured afters* recomputed, that is, the set of nodes potentially affected by the reclassification of nodes in *changed*, $\{n \mid m \in \mathit{changed} \wedge x \in \Sigma^\tau \wedge n \overset{x}{\hookrightarrow} m\}$ (lines 3 to 7). From the definition of $\hookrightarrow$, we can decompose this into $P \cup \{n \mid m \in P \wedge n \Longrightarrow m\}$ where P contains the reclassified nodes and their visible predecessors, $\mathit{changed} \cup \{n \mid m \in \mathit{changed} \wedge x \in \Sigma \wedge n \xrightarrow{x} m\}$. P is a subset of the nodes affected in the strong bisimulation sense, and can be computed using the map from nodes to their visible predecessors with the worst case run time in $O(t)$. We then compute *affected* (simultaneously updating *affected_topo*) using the map from nodes to their τ predecessors and any of the well-known graph exploration algorithms. Our implementation uses a depth-first search; to facilitate this, P is computed directly into a stack. The search takes $O(t)$ time at worst.

Next, we need to recompute the *coloured afters* for each of the nodes indexed by *affected_topo* (lines 9 to 22). To do this, we will use dynamic programming as described in Section 4.3.3. Finally, we proceed to sort and reclassify the nodes as in Section 4.2.3, recording which nodes have been reclassified in *changed*. We must also clear *affected* and *affected_topo* for the next iteration.

Figure 4.14 illustrates the important points of this algorithm on a small example.

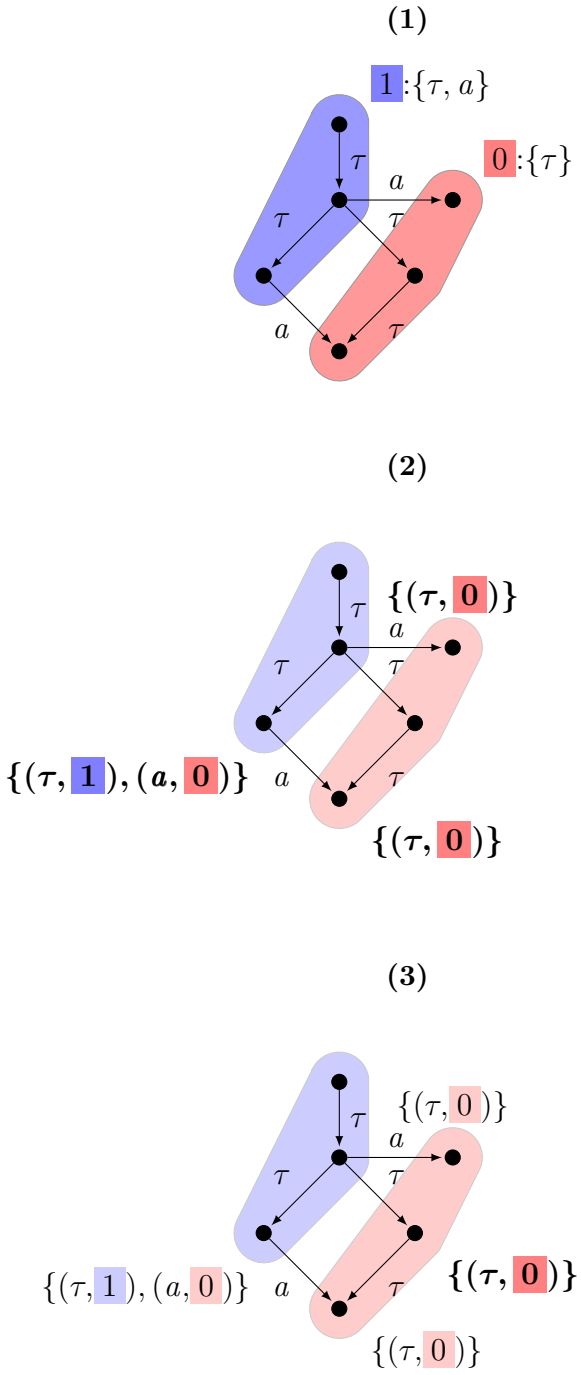


Figure 4.14: An illustration of CTIR for delay bisimulation with dynamic programming. The coloured regions indicate equivalence classes. Bold face and brighter colours emphasise newly computed *coloured afters* and equivalence classes. First, an initial partition is made based on the *initials* (step 1). Then, the *coloured afters* are computed in topological order (steps 2-5). A reclassification splits class 2 from 1, invalidating some *coloured afters* (step 6); these are struck through. The invalidated *coloured afters* are recomputed in steps 7-9. The resulting partition is stable.

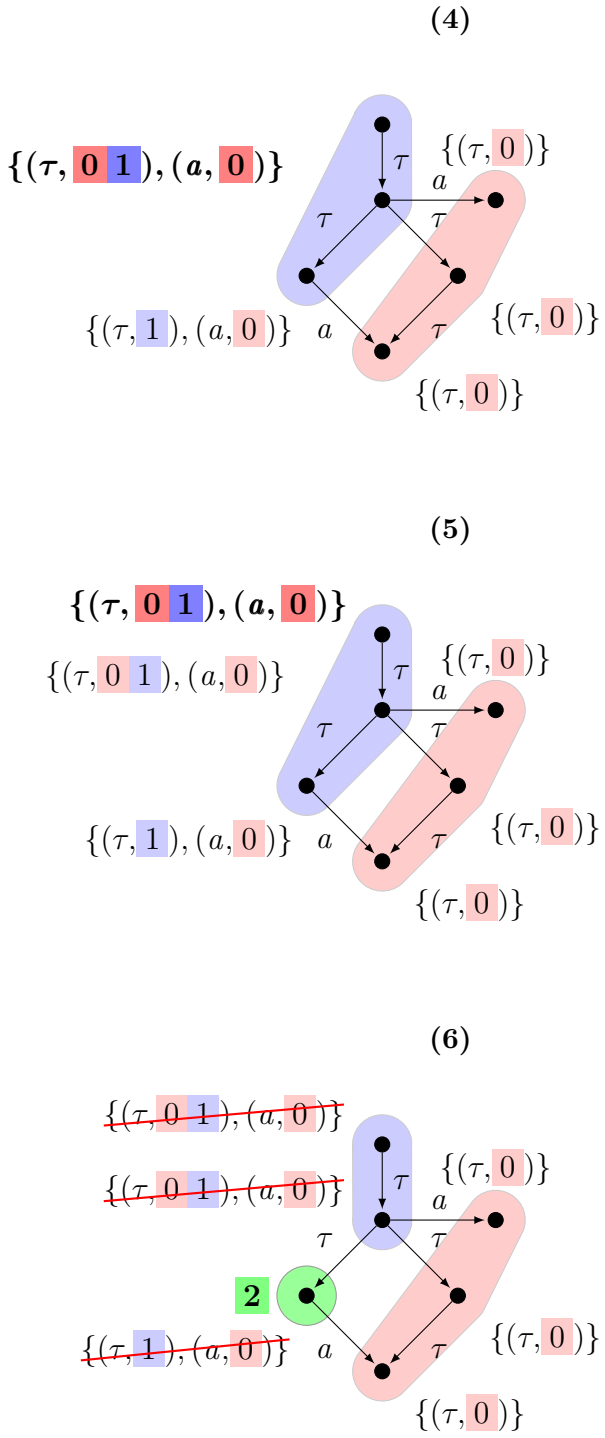


Figure 4.14: An illustration of CTIR for delay bisimulation with dynamic programming. *(Continued.)*

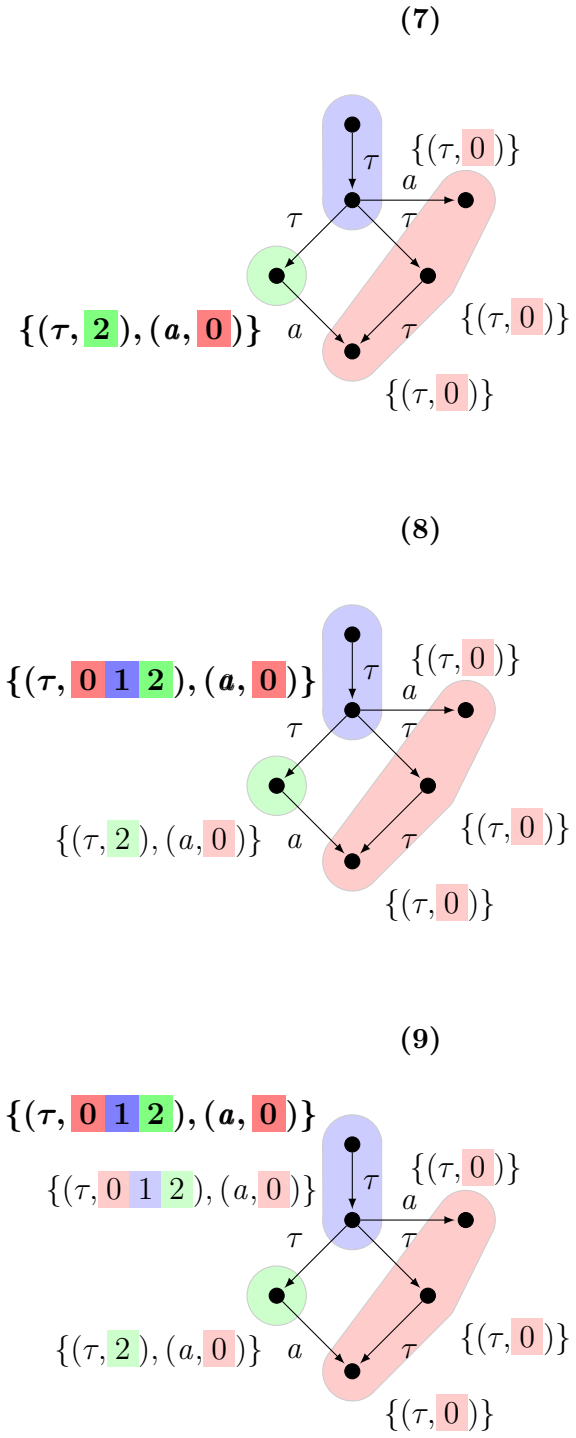


Figure 4.14: An illustration of CTIR for delay bisimulation with dynamic programming. *(Continued.)*

Construction. The output GLTS is constructed as usual.

Complexity. Factoring the cost of change tracking into the analysis for the version without change tracking, we note that the performance is still dominated by the iterated *coloured afters* computation, and is in the worst case $O(An^3c)$. In practice, change tracking should reduce the number of nodes that need their *coloured afters* recomputed at each iteration and improve the typical case.

The additional data structures required do not take up a significant amount of space. The topological sort and the bit vectors take $\Theta(n)$ space. The predecessor maps combined have no more than t entries, and will often have fewer since transitions differing only in their visible events only require one entry.

4.3.5.2 Change-Tracking DRWB

The above algorithm can be adapted to compute the maximal DRWB instead of the maximal DRDB, as shown in Figure 4.15. The iterations are adapted as follows, while everything else remains unchanged.

Iteration. We first compute the set $\{n \mid m \in \mathit{changed} \wedge x \in \Sigma^\tau \wedge n \stackrel{x}{\Longrightarrow} m\}$ of nodes whose *coloured afters* need to be recomputed. This can be decomposed into $P_1 \cup P_2 \cup P_3$, where

$$P_1 = \{n \mid m \in \mathit{changed} \wedge n \Longrightarrow m\}$$
$$P_2 = \{n \mid m \in P_1 \wedge x \in \Sigma \wedge n \stackrel{x}{\longrightarrow} m\}$$
$$P_3 = \{n \mid m \in P_2 \wedge n \Longrightarrow m\}.$$

P_1 can be computed using the map from nodes to their τ predecessors and any of the well-known graph exploration algorithms, such as a depth-first search (line 4). P_2 can be computed by iterating through P_1 and using the map from nodes to their visible predecessors (lines 5 to 8). Alternatively, computation of P_1 and P_2 can be interleaved by adding the visible predecessors of each node visited during the exploration of P_1

```
1: global toponodes, changed, affected, cafters, T^{-1}, T_τ^{-1}
2: function ComputeChangedObservedAfters((N, Σ, ⟶, Λ, λ), ρ)
3:     affected ← ∅
4:     Add all nodes reachable from changed via T_τ^{-1} to affected using, e.g., a DFS.
5:     P_2 ← ∅
6:     for each n ∈ affected do
7:         P_2 ← P_2 ∪ T^{-1}(n)
8:     end for
9:     Add all nodes reachable from P_2 via T_τ^{-1} to affected using, e.g., a DFS.
10:
11:    for each n ∈ affected_topo do   ▷ Compute nodes reachable in 0 or more τs:
12:        for each (a, M) ∈ cafters(n) do
13:            cafters(n)(a) ← ∅
14:        end for
15:        cafters(n)(τ) ← {ρ(n)}
16:        for each m ∈ {m | n ⟶^τ m} do
17:            cafters(n)(τ) ← cafters(n)(τ) ∪ cafters(m)(τ)
18:        end for
19:    end for
20:        ▷ Compute nodes reachable in 0 or more τs, 1 visible event, 0 or more τs:
21:    for each n ∈ affected_topo do
22:        for each (a, m) ∈ {(a, m) | n ⟶^a m ∧ a ≠ τ} do
23:            cafters(n)(a) ← cafters(n)(a) ∪ cafters(m)(τ)
24:        end for
25:        for each m ∈ {m | n ⟶^τ m} do
26:            for each (a, M) ∈ cafters(m), a ≠ τ do
27:                cafters(n)(a) ← cafters(n)(a) ∪ M
28:            end for
29:        end for
30:    end for
31:    return cafters
32: end function
```

Figure 4.15: The dynamic programming algorithm for DRWB with change tracking. The WBisimCTDP function itself is not listed here due to space limitations; DBisimCTDP from Figure 4.12 can be used with the obvious modifications. The global variables referenced here are analogous to those presented in Figure 4.12.

to P_2 immediately. P_3 can be computed with another depth-first search starting from P_2 (line 9). While computing these sets, *affected_topo* and *affected* should be kept up to date. We can recompute the *coloured afters* for each of the affected nodes by iterating through *affected_topo* and using the dynamic programming approach outlined in Section 4.3.4 (lines 11 to 30). Finally, we proceed to sort and reclassify the nodes as in Section 4.2.3, recording which nodes have been reclassified in *changed*. We must also clear *affected* and *affected_topo* for the next iteration.

Performance. Each of the P_i can be computed in $O(t)$ time, so the worst-case time complexity remains in $O(An^3c)$.

4.3.5.3 Single-Pass Change-Tracking DRDB

Instead of separating change tracking and the dynamic *coloured afters* computation into two passes, they can be performed in the same pass, as shown in Figure 4.16. Since the *coloured afters* need to be computed in topological order, but the affected nodes will in general not be discovered in this order, we will store the affected nodes in a min-priority queue, *affected_queue*, with each node's priority being its position in the topological order (so more downstream nodes will be pulled first). All insertions into (but not pulls from) *affected_queue* will be mirrored to the *affected* bit vector, so that the latter can be used for checking whether a node has already been seen and during the reclassification phase for determining which nodes' *coloured afters* might have changed. We do not use a *changed* bit vector in this algorithm, but we will still use it in the analysis to represent the set of nodes that changed class in the previous iteration. For change tracking, we will need a map from nodes to their predecessors under a single visible event and another map from nodes to their predecessors under a single τ. We also need to topologically sort the nodes and record a bidirectional map from nodes to their indices in the sort.

```
1: global toponodes, changed, affected, cafters, T^{-1}, T_τ^{-1}

2: function ComputeChangedDelayedAfters'((N, Σ, ⟶, Λ, λ), ρ)
3:     affected_queue ← empty min-priority queue
4:     for each n ∈ changed do
5:         affected ← affected ∪ T^{-1}(n) ∪ {n}
6:     end for
7:     affected_queue ← min-priority queue from affected
8:
9:     while affected_queue is not empty do
10:        n ← pull(affected_queue)
11:        for each (a, M) ∈ cafters(n) do
12:            cafters(n)(a) ← ∅
13:        end for
14:        cafters(n)(τ) ← {ρ(n)}                          ▷ Implicit self-loop.
15:        for each (a, m) ∈ {(a, m) | n --a--> m} do      ▷ Visible afters.
16:            cafters(n)(a) ← cafters(n)(a) ∪ {ρ(m)}
17:        end for
18:        for each m ∈ {m | n --τ--> m} do                ▷ Delayed afters.
19:            for each (a, M) ∈ cafters(m) do
20:                cafters(n)(a) ← cafters(n)(a) ∪ M
21:            end for
22:        end for
23:
24:        insert_all(affected_queue, T_τ^{-1}(n) \ affected)
25:        affected ← affected ∪ T_τ^{-1}(n)
26:    end while
27:    return cafters
28: end function
```

Figure 4.16: The single-pass dynamic programming algorithm for DRDB with change tracking. The DBisimCTDP' function itself is not listed here due to space limitations; DBisimCTDP from Figure 4.12 can be used with the obvious modifications.

Initial Approximation. The initial approximation is computed as in Section 4.3.3. All nodes are marked changed by inserting them into *affected_queue* (and correspondingly *affected*).

Iteration. As a precondition to each iteration, we expect *affected_queue* to contain all the reclassified nodes and their visible predecessors, $P = \mathit{changed} \cup \{n \mid m \in \mathit{changed} \wedge x \in \Sigma \wedge n \xrightarrow{x} m\}$ (lines 3 to 7). We then repeat the following steps until *affected_queue* is empty:

Pull a node n from *affected_queue* (line 10). We will only update *coloured afters* for nodes pulled from *affected_queue* and we will only insert nodes with higher topological indices during this iteration of *affected_queue*, so we know all the nodes downstream from n have had their *coloured afters* computed for this iteration. Therefore, compute and record n's *coloured afters* according to $\widehat{S}$ as the union of itself after τ (line 14) its *coloured afters* according to S (lines 15 to 17) and the *coloured afters* according to $\widehat{S}$ of all nodes reachable from n in exactly one τ (lines 18 to 22). Finally, insert each of the predecessors of n under τ that is not already in *affected* into *affected_queue* and *affected* (lines 24 to 25); this is the step that ensures that all nodes in $\{n \mid m \in P \wedge n \Longrightarrow m\} = \{n \mid m \in \mathit{changed} \wedge x \in \Sigma^\tau \wedge n \overset{x}{\hookrightarrow} m\}$ eventually get inserted into *affected_queue*.

By this point, *affected_queue* is empty, so all the affected nodes have had their *coloured afters* updated and have been marked in *affected*. We can therefore proceed with the reclassification stage described in Section 4.2.3. However, instead of inserting changed nodes into *changed*, we will insert them and their visible predecessors into *affected_queue*, to satisfy the precondition of the next iteration.

Construction. The output GLTS is constructed as usual.

Complexity. The *coloured afters* computation and reclassification are the same as in our other dynamic programming approaches. It is only the change tracking that is different, now that in addition to the graph traversal we have $O(n)$ pulls from and

insertions into *affected_queue*, which can be done in $O(n \log n)$ time. This does not increase the overall asymptotic time complexity from $O(An^3c)$, but can in practice impact performance (see Section 4.4.3). Because of this and the existence of the simpler two-pass algorithm, we have not developed a similarly interleaved algorithm for DRWB.

4.3.6 DRDB with the Paige-Tarjan Algorithm

We can adapt the multilabel version of the Paige-Tarjan algorithm presented in [Fer90] to compute a maximal DRDB. This will use a similar form of change tracking as that described in Section 4.3.5.1.

If the input GLTS does not have node labels, we initially assign all the nodes to the same class. Otherwise, we topologically sort the nodes and compute their node labels according to $\widehat{S}$ using dynamic programming. We then create an initial partition based on the node labels.

We leave the core of the algorithm unmodified, but instead of using the inverted transition relation of $\widehat{S}$ (which would suffer from the potentially quadratic explosion discussed in Section 4.3.2), we compute it dynamically each time it is requested. To facilitate this, we pre-compute the inverted transition relation of S (this can be done in $\Theta(t)$ time). When we need to visit the nodes in $\{m \mid m \overset{a}{\hookrightarrow} n\}$, we compute a starting set P such that $\{m \mid m \overset{a}{\hookrightarrow} n\} = \{m \mid p \in P \wedge m \Longrightarrow p\}$. If $a = \tau$, $P = \{n\}$; otherwise, $P = \{m \mid m \xrightarrow{a} n\}$. We then visit all nodes reachable from P in the inverted transition relation of S under τ using any of the known algorithms, such as depth-first search. This exploration can take time in $O(t)$.

Complexity. The added computation can increase the cost of exploring the predecessors of a node from $O(|\{m \mid m \xrightarrow{a} n\}|)$ as shown in [PT87] to $O(t)$. The cost of a single refinement by block B therefore takes $O(At|B|)$ time. Recalling from [PT87] that each node can be in at most $\log_2 n + 1$ blocks used for refinement and summing over all such blocks, we get a total run time in $O(Atn \log n)$.

4.3.7 DRWB with the Paige-Tarjan Algorithm

We can similarly adapt the multilabel version of the Paige-Tarjan algorithm presented in [Fer90] to compute a maximal DRWB. This will use a similar form of change tracking to that described in Section 4.3.5.2. We can create the initial partition by dynamically computing the labels of each node according to $\widehat{S}$ as described in Section 4.3.6.

To dynamically compute the inverted transition relation of $\widehat{S}$, we pre-compute the inverted transition relation of S. When we need to visit the nodes in $\{m \mid m \stackrel{a}{\Longrightarrow} n\}$, we note that this is equivalent to P_3, where

$$P_1 = \{m \mid m \Longrightarrow n\}$$
$$P_2 = \{m \mid p \in P_1 \wedge m \stackrel{a}{\longrightarrow} p\}$$
$$P_3 = \{m \mid p \in P_2 \wedge m \Longrightarrow p\}.$$

So, we compute P_1 using the precomputed inverted transition relation and any of the known algorithms, such a depth-first search. P_2 can be computed by iterating through P_1 and again using the inverted transition relation. Finally, P_3 can be explored using another depth-first search. The entire process can take time in $O(t)$.

Complexity Applying the same logic as in Section 4.3.6, we get a total run time in $O(Atn \log n)$.

4.4 Performance

4.4.1 Benchmark Descriptions

In the following performance tests we will compare the performances of the algorithms described throughout this chapter to each other, as well as to implementations provided by the BCG_MIN tool included in the CADP Toolbox [GLMS13].

Our primary source for example LTSs is the Very Large Transition Systems (VLTS)

Table 4.1: Parameters of the VLTS Benchmark Suite [GLMS13] used in our performance tests. The branching factor is given as an average and a range.

Name	States	Transitions	τs	$\lvert\Sigma\rvert$	Branching	
cwi_3_14	3996	14552	14551	2	3.64	[0 - 6]
vasy_18_73	18746	73043	39217	17	3.90	[1 - 6]
vasy_25_25	25217	25216	0	25216	1.00	[0 - 1]
vasy_40_60	40006	60007	20003	3	1.50	[1 - 2]
vasy_52_318	52268	318126	130752	17	6.09	[1 - 17]
vasy_65_2621	65537	2621480	0	72	40.00	[40 - 40]
vasy_66_1302	66929	1302664	117866	81	19.46	[2 - 42]
vasy_69_520	69754	520633	1	135	7.46	[0 - 35]
vasy_83_325	83436	325584	45696	211	3.90	[0 - 96]
vasy_116_368	116456	368569	263296	21	3.16	[1 - 8]
cwi_142_925	142472	925429	862298	7	6.50	[0 - 9]
vasy_157_297	157604	297000	31798	235	1.88	[0 - 48]
vasy_164_1619	164865	1619204	109910	37	9.82	[1 - 16]
vasy_166_651	166464	651168	91392	211	3.91	[0 - 96]
cwi_214_684	214202	684419	550611	5	3.20	[0 - 7]
cwi_371_641	371804	641565	445600	61	1.73	[1 - 25]
vasy_386_1171	386496	1171872	122976	73	3.03	[1 - 38]
cwi_566_3984	566640	3984157	3666614	11	7.03	[0 - 10]
vasy_574_13561	574057	13561040	0	141	23.62	[1 - 64]
vasy_720_390	720247	390999	1	49	0.54	[0 - 39]
vasy_1112_5290	1112490	5290860	0	23	4.76	[3 - 6]
cwi_2165_8723	2165446	8723465	3830225	26	4.03	[1 - 14]
cwi_2416_17605	2416632	17605592	17490904	15	7.29	[0 - 14]
vasy_2581_11442	2581374	11442382	2508518	223	4.43	[0 - 97]
vasy_4220_13944	4220790	13944372	2546649	223	3.30	[0 - 97]
vasy_4338_15666	4338672	15666588	3127116	223	3.61	[0 - 97]
vasy_6020_19353	6020550	19353474	17526144	511	3.21	[2 - 260]
vasy_6120_11031	6120718	11031292	3152976	125	1.80	[0 - 16]
cwi_7838_59101	7838608	59101007	22842122	20	7.54	[3 - 13]
vasy_8082_42933	8082905	42933110	2535944	211	5.31	[0 - 48]
vasy_11026_24660	11026932	24660513	2748559	119	2.24	[0 - 13]
vasy_12323_27667	12323703	27667803	3153502	119	2.25	[0 - 13]
cwi_33949_165318	33949609	165318222	74133306	31	4.87	[1 - 17]

Benchmark Suite[5] that is provided alongside the CADP Toolbox; some information about the LTSs it contains is given in Table 4.1. Note that some of the included LTSs do not contain any τ transitions, meaning that `dbisim` and `wbisim` cannot offer more compression than `sbisim`. We will still include them in all the tests for completeness. We have excluded those examples where the run time for all of the tested algorithms did not exceed 1 second.[6]

To ensure proper operation of FDR3, we have developed a suite of regression and feature tests containing tests generated randomly at runtime, examples from [Ros98] and [Ros10], and assorted test files. CSP processes are frequently written as compositions of smaller component processes, which in turn can be such compositions. To help mitigate the state explosion that can result from such combinations, FDR2 onwards can automatically apply compressions to component processes (those components that are not composed of other processes, and will be represented as explicit GLTSs). By default, this compression is `sbisim`, so many of the tests exercise `sbisim`. There are about 90,000 invocations of `sbisim` over the test suite, and they are a good comparison of the algorithms' performance on small component components typical in a system that does not use `sbisim` explicitly. Due to the nature of these tests, results are not available for CADP.

4.4.2 Strong Bisimulation Performance

We will now compare the performance of the `sbisim` algorithms on several examples. The test system used contains a server-grade CPU[7] and 256 GiB of RAM, running 64-bit Debian GNU/Linux 7.8. We will use the following abbreviations to refer to the algorithms in column headings:

[5]At the time of writing, the VLTS Benchmark Suite is available from `http://cadp.inria.fr/resources/vlts/`.

[6]Since FDR usage frequently involves a large number of invocations of the algorithms, even small differences in run time would add up. However, it is difficult to accurately time invocations that are too brief. We feel that 1 second is an appropriate threshold for these benchmarks.

[7]The system has two 8-core 2 GHz Intel® Xeon® E5-2650 CPUs with 20 MB of cache each. Our bisimulation algorithms are single-threaded, so the number of CPUs and cores is not significant.

NIRa Naïve Iterative Refinement using a tree-based set representation

CTIRa Change-Tracking Iterative Refinement using a tree-based set representation

NIRb Naïve Iterative Refinement using a sorted array set representation

CTIRb Change-Tracking Iterative Refinement using a sorted array set representation (used by FDR3)

PT(LL) Multilabel Paige-Tarjan algorithm using linked lists

PT(A) Multilabel Paige-Tarjan algorithm using arrays

CADP `BCG_MIN -strong` from the CADP Toolbox [GLMS13]

We can see from Table 4.2 that for most of our experiments, change-tracking iterative refinement outperforms naïve iterative refinement and the Paige-Tarjan algorithm: this is particularly noticeable for larger checks. For smaller inputs, naïve iterative refinement is sometimes slightly faster due to a smaller bookkeeping overhead, but not significantly so. The Paige-Tarjan algorithm was particularly slow on `vasy_25_25` due to its large alphabet; our iterative refinement algorithms were designed for labelled transition systems and do not suffer from the large alphabet. CADP's implementation of strong bisimulation was comparable to our change-tracking iterative refinement with sorted vectors, performing better on some examples and worse on others. We have omitted NIRa and CTIRa from the table in order to avoid cluttering it with implementation details. On average, NIRa took 2.3 times as long as NIRb and CTIRa took 1.6 times as long as CTIRb on these VLTS examples.

In the FDR3 test suite, the move from naïve to change-tracking iterative refinement affords a noticeable speedup, as evidenced by Table 4.3. Much of this speedup can be attributed to a number of outliers that take a particularly long time with naïve iterative refinement. The Paige-Tarjan algorithm is noticeably slower than both naïve and change-tracking iterative refinement, but part of this might be due to the heavy

Table 4.2: Run times of various implementations of strong bisimulation on the VLTS benchmarks in seconds.

Name	NIRb	CTIRb	PT(LL)	PT(A)	CADP
cwi_3_14	0.051	0.013	**0.010**	0.011	0.256
vasy_18_73	0.118	**0.062**	0.191	0.197	0.283
vasy_25_25	**0.022**	0.029	59.035	54.115	1.533
vasy_40_60	203.763	35.430	**2.284**	7.031	70.081
vasy_52_318	0.500	**0.392**	1.210	0.919	0.569
vasy_65_2621	**1.177**	1.436	21.275	26.563	2.642
vasy_66_1302	**0.640**	0.728	8.130	9.598	1.360
vasy_69_520	0.475	**0.410**	3.815	4.814	0.791
vasy_83_325	**0.331**	0.387	4.068	5.163	0.695
vasy_116_368	0.811	**0.752**	2.422	3.298	0.955
cwi_142_925	1.134	**0.817**	1.785	1.583	1.355
vasy_157_297	1.527	**0.307**	3.157	1.817	0.496
vasy_164_1619	1.762	1.369	5.515	3.987	**1.280**
vasy_166_651	0.894	**0.815**	7.907	11.389	0.974
cwi_214_684	6.290	**1.636**	2.915	3.529	2.440
cwi_371_641	9.112	**1.342**	4.178	4.162	2.642
vasy_386_1171	2.159	1.376	4.458	3.256	**1.193**
cwi_566_3984	6.491	**4.217**	10.234	7.834	5.588
vasy_574_13561	9.327	9.469	51.622	32.324	**8.171**
vasy_720_390	0.993	**0.413**	1.766	1.039	0.653
vasy_1112_5290	6.925	6.046	26.715	20.254	**4.093**
cwi_2165_8723	100.482	**14.547**	37.711	37.682	16.606
cwi_2416_17605	57.061	**15.115**	42.796	35.995	28.379
vasy_2581_11442	29.123	**20.982**	234.762	2438.673	28.410
vasy_4220_13944	85.000	**21.910**	267.716	734.701	26.866
vasy_4338_15666	51.796	**28.358**	332.553	1948.656	32.280
vasy_6020_19353	70.039	36.194	897.281	438.441	**22.998**
vasy_6120_11031	119.617	23.526	196.442	205.547	**15.888**
cwi_7838_59101	749.059	**115.385**	490.897	1063.371	135.400
vasy_8082_42933	130.746	73.546	416.726	280.090	**39.251**
vasy_11026_24660	361.281	50.874	444.343	847.334	**41.733**
vasy_12323_27667	303.017	56.320	509.774	1010.133	**49.498**
cwi_33949_165318	3256.559	**334.751**	962.506	1151.239	364.443
Geometric Mean	7.353	**3.236**	17.785	21.275	4.993

Table 4.3: `sbisim` timings for the FDR3 test suite. Total and worst-case runtime each algorithm.

	NIRa	CTIRa	NIRb	CTIRb	PT(LL)	PT(A)
Total	62.044	29.159	30.184	**20.313**	45.326	51.961
Worst	5.068	0.529	2.004	0.32	0.289	0.405

optimisation that our implementation of iterative refinement has gone through over the years. It should be noted, however, that the worst-case run time for the Paige-Tarjan algorithm is similar to those for change-tracking iterative refinement.

4.4.3 Divergence-Respecting Delay Bisimulation Performance

We will use the following abbreviations to refer to the various DRDB algorithms compared in column headings, with all variants of iterative refinement using sorted arrays to represent the *coloured afters*:

FDR2 Reduction to strong bisimulation

DYN Naïve Iterative Refinement with dynamic computation of *coloured afters* (used by FDR3.0-3.1)

CT-DYN Two-pass Change-Tracking Iterative Refinement with dynamic computation of *coloured afters* (used by FDR3.2 onwards)

CT-DYN1 Single-pass Change-Tracking Iterative Refinement with dynamic computation of *coloured afters*

PT Paige-Tarjan algorithm using arrays

From Table 4.4 we can see that two-pass change-tracking iterative refinement with dynamic computation of *coloured afters* is significantly faster than the alternatives on nearly all of the examples. We also note that as for strong bisimulation, the Paige-Tarjan algorithm is particularly ineffective when dealing with large alphabets, as in `vasy_25_25` since it must repeat each refinement for every event in the alphabet.

Table 4.4: Run times of various implementations of delay bisimulation on the VLTS benchmarks in seconds. A — indicates that the test failed to complete within 4 hours.

Name	FDR2	DYN	CT-DYN1	CT-DYN	PT
cwi_3_14	4.640	**0.010**	0.011	0.011	0.347
vasy_18_73	1.018	0.334	0.222	**0.204**	1.470
vasy_25_25	**0.061**	0.107	0.096	0.096	663.200
vasy_40_60	169.843	160.661	15.689	15.620	**7.097**
vasy_52_318	6.722	0.378	0.383	**0.353**	3.517
vasy_65_2621	2.960	**1.585**	1.723	1.732	63.035
vasy_66_1302	2.720	1.283	1.517	**1.154**	31.171
vasy_69_520	1.589	0.643	0.567	**0.530**	17.922
vasy_83_325	1.030	**0.433**	0.479	0.443	23.468
vasy_116_368	70.398	4.993	2.631	**2.391**	74.627
cwi_142_925	525.796	0.840	0.771	**0.717**	101.659
vasy_157_297	3.214	2.238	0.630	**0.564**	18.707
vasy_164_1619	4.060	2.903	2.197	**2.018**	14.724
vasy_166_651	3.302	0.985	1.038	**0.938**	50.885
cwi_214_684	113.378	5.116	1.956	**1.779**	28.247
cwi_371_641	96.592	2.608	2.251	**1.985**	95.357
vasy_386_1171	14.717	2.336	2.060	**1.963**	25.967
cwi_566_3984	—	4.714	4.486	**4.133**	826.634
vasy_574_13561	52.534	**13.522**	14.301	13.921	178.554
vasy_720_390	44.242	2.464	1.520	**1.412**	13.268
vasy_1112_5290	132.888	9.715	9.261	**8.667**	192.439
cwi_2165_8723	—	45.885	28.791	**26.823**	7262.560
cwi_2416_17605	—	**18.461**	20.858	19.548	—
vasy_2581_11442	730.577	29.734	23.908	**19.694**	6695.113
vasy_4220_13944	2128.400	83.883	34.915	**29.632**	4359.442
vasy_4338_15666	2257.923	51.196	35.149	**29.612**	6638.925
vasy_6020_19353	7448.062	**8.199**	9.143	8.739	811.669
vasy_6120_11031	6305.384	131.807	34.635	**30.236**	6980.648
cwi_7838_59101	—	9905.934	5214.302	**5017.498**	—
vasy_8082_42933	—	117.119	96.233	**89.629**	6268.539
vasy_11026_24660	—	442.642	94.820	**80.174**	—
vasy_12323_27667	—	434.968	108.348	**89.650**	—
cwi_33949_165318	—	1198.680	663.610	**592.355**	—
Geometric Mean	—	7.937	5.211	**4.759**	—

4.4.4 Divergence-Respecting Weak Bisimulation Performance

We will use the following abbreviations to refer to the various DRWB algorithms compared in column headings, with both variants of iterative refinement using sorted arrays to represent the *coloured afters*:

DYN Naïve Iterative Refinement with dynamic computation of *coloured afters*

CT-DYN Two-pass Change-Tracking Iterative Refinement with dynamic computation of *coloured afters* (used by FDR3.2 onwards)

PT Paige-Tarjan algorithm using arrays

CADP-nobr `BCG_MIN -observational -class` from the CADP Toolbox [GLMS13]

CADP-br `BCG_MIN -observational` from the CADP Toolbox [GLMS13]

`BCG_MIN -observational` by default applies branching bisimulation before applying weak bisimulation; this is what CADP-br measures. This behaviour can be disabled with the `-class` option, allowing us to measure the performance of the weak bisimulation directly; we denote this CADP-nobr.

From Table 4.5 we can see that change-tracking iterative refinement with dynamic computation of *coloured afters* is significantly faster than the alternatives that rely solely on weak bisimulation on nearly all of the examples. CADP's branching bisimulation followed by weak bisimulation was found to be slower or faster depending on the example.

Table 4.5: Run times of various implementations of weak bisimulation on the VLTS benchmarks in seconds. A — indicates that the test failed to complete within 4 hours.

Name	DYN	CT-DYN	PT	CADP-nobr	CADP-br
cwi_3_14	**0.010**	0.011	0.542	2.632	0.244
vasy_18_73	0.448	**0.279**	5.125	3.615	0.742
vasy_25_25	**0.092**	0.094	1558.280	1.134	1.468
vasy_40_60	170.634	15.734	**7.222**	109.133	115.415
vasy_52_318	0.424	**0.383**	9.032	7.152	0.806
vasy_65_2621	**1.605**	2.069	64.023	2.543	5.196
vasy_66_1302	1.442	**1.364**	65.053	9.512	8.370
vasy_69_520	0.783	**0.597**	21.125	6.573	2.001
vasy_83_325	0.492	**0.471**	69.043	12.762	0.997
vasy_116_368	5.990	**3.070**	410.344	221.241	48.121
cwi_142_925	**0.736**	0.767	306.684	290.736	1.098
vasy_157_297	2.513	**0.649**	37.907	45.589	0.694
vasy_164_1619	3.309	2.265	40.964	11.796	**1.947**
vasy_166_651	1.131	**1.041**	202.888	47.152	1.369
cwi_214_684	5.119	**2.177**	120.182	500.183	2.715
cwi_371_641	3.448	2.731	1913.489	413.884	**1.602**
vasy_386_1171	2.503	2.049	125.959	85.951	**1.452**
cwi_566_3984	5.358	4.531	3914.916	—	**4.085**
vasy_574_13561	14.624	14.749	210.813	**8.404**	8.618
vasy_720_390	2.541	**1.458**	14.330	195.721	1.468
vasy_1112_5290	10.895	9.523	238.637	**4.240**	4.375
cwi_2165_8723	80.916	41.494	—	7424.895	**13.869**
cwi_2416_17605	18.730	20.600	—	—	**16.691**
vasy_2581_11442	39.954	**23.773**	—	—	31.457
vasy_4220_13944	111.596	**37.217**	—	—	4148.586
vasy_4338_15666	64.766	**34.958**	—	—	38.171
vasy_6020_19353	**8.588**	9.167	—	—	12.011
vasy_6120_11031	149.394	33.287	—	—	**21.602**
cwi_7838_59101	12422.744	**6499.632**	—	—	9164.557
vasy_8082_42933	126.767	98.128	—	—	**42.469**
vasy_11026_24660	538.958	**91.665**	—	—	607.453
vasy_12323_27667	533.449	**103.169**	—	—	778.553
cwi_33949_165318	2665.535	1099.680	—	—	**259.913**
Geometric Mean	9.232	**5.518**	—	—	10.451

4.4.5 Other Compressions

It is interesting to compare `dbisim` and `wbisim` with alternative compressions. Prior to their introduction in FDR2.94 and 3.0, the most widely used compression was `sbisim(diamond(`P`))`, which we will call `sbdia`. In all the following examples `sbdia` is valid. Other tools also use divergence-respecting *branching* bisimulation due to the existence of a fast algorithm for computing it.

We examined the performance and effectiveness of `dbisim` and `sbdia` on the *bully* algorithm (the FDR implementation is outlined in Section 14.4 of [Ros10]) with 5 processors and an implementation of Lamport's bakery algorithm (Section 18.5 of [Ros10]) with either 3 or 4 threads and integers drawn from the fixed range 0 to 7.[8] These are typical examples composed of a variable number of parallel processes, with many τs and symmetry that can be reduced by either `dbisim` or `sbdia`. We compressed these processes *inductively*[9] (as described in Section 8.8 of [Ros10]); that is, we added them to the composition one at a time, compressing at every step. This is a common technique that allows a large portion of the system to be compressed while keeping each compression's inputs manageable. Table 4.6 shows that `sbdia` runs faster than `dbisim` and Table 4.7 shows that it is more effective at reducing state counts, but can add transitions, whereas `dbisim` cannot by design.

Tables 4.8 and 4.9 compare the effectiveness of `wbisim`, `dbisim`, branching bisimulation, and `sbdia` on the VLTS benchmarks. Table 4.10 compares their run times, as well as `sbisim`'s. We have found that `sbisim` is frequently slower than the other compressions despite being less effective.

[8] The example files are available from the author's website.

[9] We used inductive compression to increase the time spent on the compressions. This is not necessarily the most efficient approach to checking these systems in FDR.

Table 4.6: Timings with no compression, `dbisim`, and `sbdia`. "Raw" indicates the size before compression.

Problem	Compilation Time (s)			Exploration Time (s)		
	Raw	`dbisim`	`sbdia`	Raw	`dbisim`	`sbdia`
`bully`	0.06	15.82	12.81	1.12	0.24	0.42
`bakery.3`	0.22	0.26	0.27	4.56	0.83	1.25
`bakery.4`	0.37	8.49	6.59	4040.32	0.53	0.28

Table 4.7: State and transition counts with no compression, `dbisim`, and `sbdia`. "Raw" indicates the size before compression.

Problem	States			Transitions		
	Raw	`dbisim`	`sbdia`	Raw	`dbisim`	`sbdia`
`bully`	492,548	140,776	105,701	3,690,716	1,280,729	3,872,483
`bakery.3`	8,197,011	29,752	17,787	24,544,801	85,217	64,283
`bakery.4`	3,750,599,509	1,439,283	716,097	14,971,663,463	5,327,436	3,408,420

Table 4.8: A comparison of the state counts resulting from different compressions.

Problem	**Raw**	`w/dbisim`	`branch`	`sbdia`
`cwi_3_14`	3996	2	2	2
`vasy_18_73`	18746	2326	2326	**954**
`vasy_25_25`	25217	25217	25217	25217
`vasy_40_60`	40006	20003	20003	20003
`vasy_52_318`	52268	66	4593	**28**
`vasy_65_2621`	65537	65536	65536	65536
`vasy_66_1302`	66929	51128	51128	51128
`vasy_69_520`	69754	69753	69753	69753
`vasy_83_325`	83436	42195	42195	42195
`vasy_116_368`	116456	17641	22398	**616**
`cwi_142_925`	142472	19	23	**10**
`vasy_157_297`	157604	3038	3038	3038
`vasy_164_1619`	164865	992	992	**512**
`vasy_166_651`	166464	42195	42195	42195
`cwi_214_684`	214202	450	603	**222**
`cwi_371_641`	371804	2134	6033	**1496**
`vasy_386_1171`	386496	71	71	71
`cwi_566_3984`	566640	128	198	**21**
`vasy_574_13561`	574057	3577	3577	3577
`vasy_720_390`	720247	3292	3292	**3277**
`vasy_1112_5290`	1112490	265	265	265
`cwi_2165_8723`	2165446	**4256**	**4256**	4701
`cwi_2416_17605`	2416632	730	730	**2**
`vasy_2581_11442`	2581374	704737	704737	704737
`vasy_4220_13944`	4220790	1185975	1186266	**483404**
`vasy_4338_15666`	4338672	704737	704737	704737
`vasy_6020_19353`	6020550	256	256	256
`vasy_6120_11031`	6120718	2505	2505	2505
`cwi_7838_59101`	7838608	61233	62031	**36972**
`vasy_8082_42933`	8082905	290	290	290
`vasy_11026_24660`	11026932	775578/775618	775618	**637639**
`vasy_12323_27667`	12323703	876944	876944	**719324**
`cwi_33949_165318`	33949609	**12463**	**12463**	15121

Table 4.9: A comparison of the transition counts resulting from different compressions.

Problem	**Raw**	`wbisim`	`sbdia`
`cwi_3_14`	14552	1	1
`vasy_18_73`	73043	9751	**5727**
`vasy_25_25`	25216	25216	25216
`vasy_40_60`	60007	40004	40004
`vasy_52_318`	318126	333	**120**
`vasy_65_2621`	2621480	2621440	2621440
`vasy_66_1302`	1302664	**1018692**	1505446
`vasy_69_520`	520633	520632	520632
`vasy_83_325`	325584	197200	197200
`vasy_116_368`	368569	72955	**1987**
`cwi_142_925`	925429	37	**16**
`vasy_157_297`	297000	12095	12095
`vasy_164_1619`	1619204	3456	**1408**
`vasy_166_651`	651168	197200	197200
`cwi_214_684`	684419	1546	**1372**
`cwi_371_641`	641565	5634	**5100**
`vasy_386_1171`	1171872	108	108
`cwi_566_3984`	3984157	523	**50**
`vasy_574_13561`	13561040	16168	16168
`vasy_720_390`	390999	116910	**116498**
`vasy_1112_5290`	5290860	1300	1300
`cwi_2165_8723`	8723465	**20880**	87575
`cwi_2416_17605`	17605592	2899	**14**
`vasy_2581_11442`	11442382	3972600	3972600
`vasy_4220_13944`	13944372	6862722	**3420840**
`vasy_4338_15666`	15666588	3972600	3972600
`vasy_6020_19353`	19353474	510	510
`vasy_6120_11031`	11031292	5358	5358
`cwi_7838_59101`	59101007	**464102**	1369417
`vasy_8082_42933`	42933110	680	680
`vasy_11026_24660`	24660513	2454736	**1993745**
`vasy_12323_27667`	27667803	2780022	**2251773**
`cwi_33949_165318`	165318222	**71466**	500580

Table 4.10: A comparison of the run times of different compressions.

Name	`wbisim`	`dbisim`	`sbdia`	`branch`	`sbisim`
`cwi_3_14`	0.011	0.011	**0.003**	0.069	0.013
`vasy_18_73`	0.279	0.204	0.180	0.167	**0.062**
`vasy_25_25`	0.094	0.096	0.050	1.136	**0.029**
`vasy_40_60`	15.734	15.620	**11.723**	70.846	35.430
`vasy_52_318`	0.383	**0.353**	0.546	0.492	0.392
`vasy_65_2621`	2.069	1.732	2.271	2.541	**1.436**
`vasy_66_1302`	1.364	1.154	1.588	1.355	**0.728**
`vasy_69_520`	0.597	0.530	0.588	0.881	**0.410**
`vasy_83_325`	0.471	0.443	**0.308**	0.544	0.387
`vasy_116_368`	3.070	2.391	**0.233**	0.819	0.752
`cwi_142_925`	0.767	0.717	**0.332**	0.729	0.817
`vasy_157_297`	0.649	0.564	0.445	0.472	**0.307**
`vasy_164_1619`	2.265	2.018	**0.847**	1.443	1.369
`vasy_166_651`	1.041	0.938	**0.682**	0.922	0.815
`cwi_214_684`	2.177	1.779	**0.353**	1.566	1.636
`cwi_371_641`	2.731	1.985	6.465	**1.336**	1.342
`vasy_386_1171`	2.049	1.963	1.605	**1.258**	1.376
`cwi_566_3984`	4.531	4.133	**2.529**	3.665	4.217
`vasy_574_13561`	14.749	13.921	14.484	**8.325**	9.469
`vasy_720_390`	1.458	1.412	0.417	0.850	**0.413**
`vasy_1112_5290`	9.523	8.667	8.346	**4.163**	6.046
`cwi_2165_8723`	41.494	26.823	116.233	**12.907**	14.547
`cwi_2416_17605`	20.600	19.548	**3.480**	14.604	15.115
`vasy_2581_11442`	23.773	19.694	**10.129**	23.388	20.982
`vasy_4220_13944`	37.217	29.632	**16.541**	28.392	21.910
`vasy_4338_15666`	34.958	29.612	**18.876**	30.091	28.358
`vasy_6020_19353`	9.167	8.739	**7.892**	12.107	36.194
`vasy_6120_11031`	33.287	30.236	**16.981**	20.673	23.526
`cwi_7838_59101`	6499.632	5017.498	1801.493	145.481	**115.385**
`vasy_8082_42933`	98.128	89.629	78.149	**42.434**	73.546
`vasy_11026_24660`	91.665	80.174	54.986	**48.827**	50.874
`vasy_12323_27667`	103.169	89.650	60.724	**55.878**	56.320
`cwi_33949_165318`	1099.680	592.355	9102.699	**254.545**	334.751
Geometric Mean	5.518	4.759	3.483	4.183	**3.236**

4.5 Conclusions

We have presented a number of GLTS compression algorithms, including novel algorithms for computing the maximal delay and weak bisimulation. We have shown that explicitly constructing a τ-closed transition relation for weak bisimulations, the current state of the art, is prohibitively memory-intensive and provided an efficient alternative based on dynamic programming that is particularly effective when used in conjunction with change-tracking iterative refinement.

CADP's attempt to reduce this potential explosion by applying branching bisimulation reduction first, thus reducing the size of the input to weak bisimulation, was effective in some cases, but not others. Since our approach is able to cope with large transition systems without requiring them to be pre-compressed by a more time-efficient compression we believe it to be a useful contribution.

Change-tracking iterative refinement algorithm for `sbisim` offered a significant improvement over the naïve iterative refinement used by FDR2, supporting the conclusions of [BO05]. It outperformed the Paige-Tarjan algorithm as well; in particular, we found the multilabel Paige-Tarjan algorithm of [Fer90] to not be tractable for systems with large alphabets.

Comparing `dbisim` and `wbisim`, we have noticed that they produce identical output on nearly all of the examples we have tested and differ by only a few states when they do differ. They also tend to exhibit similar run times, with `wbisim` always slightly slower since it has to do strictly more work. Divergence-respecting branching bisimulation frequently produced identical output to `dbisim`, but was on some examples much less effective (for instance, compressing `vasy_52_318` from 52268 to 4593 states versus 66 states for `dbisim`). While branching bisimulation was usually faster than `dbisim`, with the exception of one VLTS example, the difference was not as striking as we had expected – roughly a factor of 2. The comparison between `dbisim` and `sbdia` was rather more varied: while the difference is not nearly as large as in FDR2.94 and there are a number of examples where `dbisim` is significantly faster than `sbdia` (most notably, 10 minutes against 150 on the largest VLTS example we tried), there are still

many examples where `sbdia` is the better choice.

We were most surprised to find that `dbisim` was occasionally faster than `sbisim`, despite offering more compression. In fact, the surprising speed is *because* of the higher compression, since the number of classes affects both the amount of work that needs to be done on each iteration and the number of iterations.

Related Work

On-the-fly τ-closure reduction [Mat05] is related to our *coloured afters* computation for DRDB. Mateescu's approach initiates a depth-first search from each of the nodes under consideration. Our method is able to avoid much of the overlapping work these DFSs might do by topologically sorting *all* the nodes and using dynamic programming. This is not possible for on-the-fly verification since the nodes are discovered as the algorithm is running, and in particular the most downstream nodes (which we need process first) are the last to be discovered. However, it would be interesting to consider whether either of the algorithms could incorporate some ideas from the other.

The strong bisimulation algorithm in [DPP04] is an improvement of the Paige-Tarjan algorithm that uses set-theoretic *rank functions* to select splitters more intelligently. By doing so they are able to achieve a two-fold speedup over the Paige-Tarjan algorithm, but more interestingly, their algorithm uses an initial partition that does not require the entire graph to be in memory at the same time. This enables scaling to larger systems, as well as opening up some potential for distributing the work, and it would be interesting to see if some of their ideas could be incorporated into Change-Tracking Iterative Refinement.

Blom and Orzan investigated the possibility of parallelising iterative refinement in [BO05]. They demonstrated implementations that obtained a nearly linear speedup on naïve iterative refinement, but a much smaller speedup on change-tracking iterative refinement. Moreover, the speedup due to change tracking was greater than the speedup from the parallelisation. We have confirmed their findings with our own parallel implementations of some of the algorithms. Since FDR3 is able to run multiple

(single-threaded) compressions simultaneously on separate threads, we have not looked further into parallelising individual compressions.

In [BvdP09], Blom and van de Pol introduce the concept of *inductive signatures* which allow the *afters* under certain events to be coloured with respect to the *current* partition instead of the *previous* one. This allows for faster convergence, but requires a *well-founded* subset of the alphabet (such that the transition set restricted to these events has no cycles); they achieve this for branching bisimulation by removing τ loops and using $\{\tau\}$ as the subset. Since this is also possible for DRDB and DRWB, it would be interesting to investigate, though it would require a data structure that allows states to be reclassified before their entire block's *coloured afters* have been recomputed (for example, a hash table).

Another common notion of bisimulation is *branching bisimulation* [vGW96]. It is coarser than strong bisimulation but finer than delay bisimulation, though as our results in Section 4.4 demonstrate, it often gives the same results as the latter. A branching bisimulation must satisfy the following, as well as the usual divergence and node label clauses if relevant:

$$\forall\, n_1, n_2, m_1 \in N \cdot \forall\, x \in \Sigma \cdot n_1\ R\ m_1 \wedge n_1 \stackrel{x}{\Longrightarrow} n_2$$
$$\Rightarrow \quad \exists m_2, m_3, m_4 \cdot m_1 \Longrightarrow m_2 \xrightarrow{x} m_3 \Longrightarrow m_4 \wedge n_1\ R\ m_2 \wedge n_2\ R\ m_3 \wedge n_2\ R\ m_4$$

$$\forall\, n_1, n_2, m_1 \in N \cdot n_1\ R\ m_1 \wedge n_1 \Longrightarrow n_2 \quad \Rightarrow \quad n_2\ R\ m_1$$

The main motivation for introducing branching bisimulation was that it more accurately captures the internal structure of a process than either delay or weak bisimulation, preserving information about possible but not-taken actions along a trace, which is necessary for branching-time semantics. However, CSP traditionally only uses linear observations as described in [Ros10] in order to preserve certain properties such as the distributivity of all CSP operators over $\sqcap$. These allow only limited information about not-taken actions to be observed, and in particular, events available from *unstable* states (those with outgoing τs) cannot be observed unless they are taken, even though branching bisimulation preserves information about them. Indeed, DRWB is

sufficiently fine for any finite observation model of CSP. Branching bisimulation has been made particularly attractive by Groote and Vaandrager's [GV90] $O(nt)$ algorithm for computing it, which was both faster and more memory-efficient than the algorithms previously existing for computing maximal weak bisimulations. However, as we have shown in Section 4.4, our new algorithms generally perform on par with current implementations of branching bisimulation.

Chapter 5

Conclusions

We have proposed a number of approaches to mitigating the *state space explosion* problem in the context of the CSP model checker FDR. We illustrated their effectiveness on a number of case studies.

In Chapter 3, we presented a *lazy compiler* for a subset of CSP_M that is particularly prone to generating systems with large numbers of unexplored states. It was previously necessary to avoid this subset of the language due to practical considerations, forcing users to search for clever ways of expressing the same systems using constructs better suited for FDR. We demonstrated that our lazy compiler allows the natural formulations of subsystems relying on these constructs to be explored efficiently while retaining the full generality of CSP_M for other subsystems.

To complement the lazy compiler which allows in some cases a more efficient generation of component processes, we next explored approaches to combine these components into a full system more efficiently. In Chapter 4, we presented a number of GLTS *compression* algorithms which allow symmetries within component processes to be exploited in order to reduce the sizes of their state spaces. While this is computationally intensive, it results in a multiplicative reduction to the size of state space of the resulting system and can be quite effective.

The presented compressions included novel algorithms for computing the maximal *divergence-respecting delay bisimulation* and *divergence-respecting weak bisimulation*

relying on *dynamic programming* to construct an implicit representation of the τ-closed transition relation. We showed this to greatly reduce memory usage. We explored the use of *change tracking* in the iterative refinement algorithm used to compute the maximal *strong bisimulation* and used as a basis for the delay and weak bisimulation algorithms. Change tracking allows for some redundant work to be eliminated, and we found it to greatly improve performance for all of the bisimulation compressions we looked at.

Bibliography

[AGL+12] P. Armstrong, M. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Recent developments in FDR. In *Computer Aided Verification*, pages 699–704. Springer, 2012.

[ALOR12] P. Armstrong, G. Lowe, J. Ouaknine, and A. W. Roscoe. Model checking Timed CSP. In *Proceedings of HOWARD (Festschrift for Howard Barringer)*, 2012.

[BBC08] BBC. Microsoft Zune affected by 'bug'. `http://news.bbc.co.uk/1/hi/technology/7806683.stm`, 2008. Accessed: 2013-10-04.

[BGRR14] A. Boulgakov, T. Gibson-Robinson, and A. W. Roscoe. Computing maximal bisimulations. In *International Conference on Formal Engineering Methods*, pages 11–26. Springer, 2014.

[BGRR16] A. Boulgakov, T. Gibson-Robinson, and A. W. Roscoe. Computing maximal weak and other bisimulations. *Formal Aspects of Computing*, 28(3):381–407, 2016.

[BH03] G. Broadfoot and P. Hopcroft. Analytical software design. *Technical report, Oxford University Computing Laboratory, Verum*, 2003.

[BO02] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Electronic Notes in Theoretical Computer Science*, 68(4):523–538, 2002.

[BO05] S. Blom and S. Orzan. Distributed state space minimization. *International Journal on Software Tools for Technology Transfer*, 7(3):280–291, 2005.

[BvdP09] S. Blom and J. van de Pol. Distributed branching bisimulation minimization by inductive signatures. In Proceedings 8th International Workshop on *Parallel and Distributed Methods in Verification,* Eindhoven, The Netherlands, 4th November 2009, volume 14 of *Electronic Proceedings in Theoretical Computer Science*, pages 32–46. Open Publishing Association, 2009.

[CE81] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[CGK+13] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. de Vink, W. Wesselink, and T. Willemse. An overview of the mCRL2 toolset and its recent advances. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[DLLF+16] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[DPP04] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1):221–256, 2004.

[ECM01] *Standard ECMA-335: The Common Language Infrastructure*. ECMA, 2001.

[ECM06] *Standard ECMA-334: C# Language Specification*. ECMA, 2006.

[EH86] E. Emerson and J. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[Fäh10] M. Fähndrich. Static verification for code contracts. In *International Static Analysis Symposium*, pages 2–5. Springer, 2010.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2):219–236, 1990.

[Flo62] R. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[GLMS13] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[God95] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. PhD thesis, Université de Liège, 1995.

[GRABR15] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3: A parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015.

[GV90] J. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In Michael S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer Berlin Heidelberg, 1990.

[HJG08] G. Holzmann, R. Joshi, and A. Groce. Swarm verification. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6. IEEE Computer Society, 2008.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*, volume 178. Prentice-Hall Englewood Cliffs, 1985.

[Hol98] G. Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3):289–307, 1998.

[Hol04] G. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HP99] G. Holzmann and A. Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[KS83] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 228–240, New York, NY, USA, 1983. ACM.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat02] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Lei10] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[Mat05] R. Mateescu. On-the-fly state space reductions for weak equivalences. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 80–89. ACM, 2005.

[MG01] E. Meijer and J. Gough. Technical overview of the Common Language Runtime. *language*, 29:7, 2001.

[Mil81] R. Milner. A modal characterisation of observable machine-behaviour. In *CAAP'81*, pages 25–34. Springer, 1981.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Berlin Heidelberg, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[POR09] H. Palikareva, J. Ouaknine, and A. W. Roscoe. Faster FDR counterexample generation using SAT-solving. *Electronic Communications of the EASST*, 23, 2009.

[Pra81] V. Pratt. A decidable mu-calculus: Preliminary report. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 421–427. IEEE, 1981.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PU96] I. Phillips and I. Ulidowski. Ordered SOS rules and weak bisimulation. *Theory and Formal Methods*, 1996.

[RAH12] A. W. Roscoe, P. Armstrong, and P. Hopcroft. Fairness analysis through priority. 2012.

[RB98] A. W. Roscoe and P. Broadfoot. Proving security protocols with model checkers by data independence techniques. In *Proceedings of CSFW 1998*. IEEE Press, 1998.

[RG97] A. W. Roscoe and M. Goldsmith. The perfect spy for model-checking crypto-protocols. In *Proceedings of DIMACS workshop on the design and formal verification of crypto-protocols*, 1997. `http://dimacs.rutgers.edu/workshops/program2/program.html`.

[RGG+95] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical compression for model-checking CSP, or How to check 10^{20} dining philosophers for deadlock. In *Proceedings of TACAS*. BRICS, 1995.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Springer, 1998.

[Ros09] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. *The Journal of Logic and Algebraic Programming*, 78(3):163–190, 2009.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[Ros11] A. W. Roscoe. On the expressiveness of CSP. Technical report, Department of Computer Science, University of Oxford, 2011.

[Ros15] A. W. Roscoe. The expressiveness of CSP with priority. In *Proceedings of MFPS*, 2015.

[RR88] A. W. Roscoe and G. M. Reed. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. W. Roscoe. *The modelling and analysis of security protocols: the CSP approach*. Addison-Wesley Professional, 2001.

[San96] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta informatica*, 33(1):69–97, 1996.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[Tar76] R. E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

[Uni13] University of Oxford. *libcspm*, 2013. `https://github.com/tomgr/libcspm`.

[vGW96] R. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, May 1996.

[WHH+06] R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref–a symbolic bisimulation tool box. In *Automated Technology for Verification and Analysis*, pages 477–492. Springer, 2006.