UNIVERSITY OF SOUTHAMPTON

Expressive and efficient bounded model checking of concurrent software

by

Jeremy Morse

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Physical Sciences and Engineering
Electronics and Computer Science

April 2015

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

EXPRESSIVE AND EFFICIENT BOUNDED MODEL CHECKING OF CONCURRENT SOFTWARE

by Jeremy Morse

To improve automated verification techniques for ANSI-C software, I examine temporal logics for describing program properties, and techniques for increasing the speed of program verification, for both single threaded and concurrent programs, based on the model checker ESBMC. A technique for evaluating LTL formulae over finite program traces is proposed and evaluated over a piece of industrial software and a suite of benchmarks, with favourable results. Efficient formulations of the model checking problem for SMT solvers are evaluated, and the performance of different solvers compared. Finally a number of optimisations for concurrent program verification not previously applied to symbolic software model checking are evaluated, resulting in an order of magnitude performance improvement over ESBMCs prior and already internationally competitive performance.

Acknowledgements

The work in this thesis would not have been possible without the continuous support of my supervisory team, Denis Nicole and Bernd Fischer – in particular, Fisch's continued help even once he had left the University. I also appreciate the pleasent and relaxed working environment created by my fellow doctoral candidates Tristan Aubrey-Jones, Neville Grech, Meng Tian, Owen Stephens and Max Barraclough. Their willingness to help sound out ideas, or even just to listen to stories of technology induced misery, greatly soothed my nerves.

My work was done in collaboration with a wider community than just the University—my predecessor Lucas Cordeiro and successor Mikhail Ramalho have both worked tirelessly on ESBMC, along with their colleagues in Manaus, helping it transform into a maturing code base that hopefully will have a bright future.

Finally, I would not have achieved anything at all without the love and support of my family, which propelled me onwards even when everything else failed.

To my brother, whose own annoyance with academia and the PhD process never prevented him from providing constructive feedback and good suggestions

Contents

List of Figures

List of publications

Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Context-bounded model checking of ltl properties for ansi-c software. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 302–317, Berlin, Heidelberg, 2011. Springer-Verlag.

Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. Context-bounded model checking with esbmc 1.17. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 534–537, Berlin, Heidelberg, 2012. Springer-Verlag.

Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Handling unbounded loops with esbmc 1.20. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 619–622, Berlin, Heidelberg, 2013. Springer-Verlag.

Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Esbmc 1.22 (competition contribution). In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'14, pages 405–407, Berlin, Heidelberg, 2014. Springer-Verlag.

Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Applying symbolic bounded model checking to the 2012 RERS greybox challenge. *STTT*, 16(5):519–529, 2014.

Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Model checking LTL properties over ANSI-C programs with bounded traces. *Software and System Modeling*, 14(1):65–81, 2015.

Chapter 1

Introduction

When systems are designed, engineering processes demand that a validated set of criteria be available to evaluate the design, *verifying* that it is is correct and meets the requirements of the application. Examples of this include verifying that a bridge will support its maximum design load, operating a high power circuit for a long period[1] to verify its robustness, or studying a piece of software's source code and deciding whether it will produce the required output for all given inputs.

As technology becomes increasingly common in the modern world, the correct operation of such technology accordingly becomes more important. In addition, technology is used to perform ever more complex tasks. Verifying that the docking system on a space shuttle operates correctly, or that systems in a nuclear plant always fail safe, are clear examples where lives are at risk and thus those systems require verification. Less prominent systems, however, can still be significantly disruptive and cause economic damage; an "off by one" error in Microsoft's Windows CE product[2] caused hundreds of thousands of Zune MP3 players to enter infinite loops on one day every leap year [141], while a similar fault in Apple's iOS software prevented timer alarms from sounding at the start of 2013. Such faults are non-fatal and may be insignificant in impact to the operator of the system, but are highly undesirable if they disrupt millions of people. Another example is the recent "Heartbleed" software vulnerability [3], which has caused an unknown amount of damage, and at the very least caused millions of people to change passwords.

The economic cost of verifying systems is, however, significant, and businesses cannot always afford to rigorously verify their products. In response, academia must produce effective technology to verify systems using a feasible amount of resources, and that delivers a reasonable assurance of system correctness.

[1] A "soak test"

[2] Whereby the programmers had assumed, when calculating the current year, that all years had 365 days, without considering leap years

This thesis explores the use of *automated verification* to deliver verification for software systems, with a particular focus on model checking to verify software using purely automatic processes. I first address several existing verification techniques to frame the chosen solution, before describing the structure of the rest of this thesis.

1.1 Testing

Verification often takes the form of a series of tests. The acceptance criteria of many projects are for the product to correctly perform certain tasks, and the only way to know this for certain is to actually perform those tasks, through testing. To help with this procedure, software projects tend to have *test suites*: collections of independent tests. Crucially, test suites can be applied automatically to the codebase, allowing engineers to verify the code they have written without individual effort, and during the development cycle.

Testing, however, is not a comprehensive solution to the existence of errors, or without costs. The failure of a test does not always indicate that incorrect behaviour is occurring, only that a behavioural change has occurred, and failures that are actual incorrect behaviours still require the engineer to analyse the system further to discover the cause. A significant amount of knowledge about the system, and imagination regarding ways it could fail, are required to write tests; test suites must also be maintained in step with the development of the system to avoid incompatibilities. This in itself leads to great expenditure of engineering time [43, 30].

Testing does not scale gracefully to verify large pieces of software. The amount of time and resources required to perform tests grows linearly with the number of tests, while the complexity of systems, the number of interacting components, and the range of inputs, ensures that the quantity of tests required to verify every aspect of the system grows exponentially.

An alternative to testing all inputs and all interactions is instead to test all lines of code. Test generation tools such as KLEE [48] can generate such tests to ensure high *coverage* of the test suite. This approach does not verify every single path through the program though, and in the presence of concurrency it becomes difficult to control the path taken by each particular test.

Despite its limitations, testing forms the backbone of verification for major projects today. The SQLite SQL library, for example, has several million tests for its codebase, consisting of many times more lines of code than the actual code base itself [4]. Testing remains difficult though, and it does not provide a complete method of finding new faults. Attention thus turns to the possibility of verifying the correct operation of designs without the need for exhaustive or specially written tests.

1.2 Properties

When attempting to automatically verify that a design is correct, a significant hurdle is deciding *what* correctness means. The design engineer has a knowledge of the correct behaviour of the system, but this must be expressed in a manner that can be understood by tools, and applied to the model of the program which the tools create. The solution to this is the creation of program *properties*. A property is some invariant of the program that must always hold. If it does not always hold (it is *violated*), then the program is incorrect in some manner, such as performing an illegal operation or producing the wrong output.

Properties of a program can be expressed about its state, perhaps that a variable must always hold a certain value, or as behaviour of the program that must always be preserved, such as always responding to a request. Properties may be negative as well as positive, i.e., that some behaviour must *never* occur.

Numerous kinds of properties exist; programming languages themselves have built in properties, stating that a program only operates correctly if its actions are well defined by the language, such as accesses to arrays always being in bounds, or pointer accesses never dereferencing a `null` reference. Engineers can state their own properties regarding program operation too: commonly by making *assertions* about current state during the course of program execution.

More elaborate properties can be expressed too. The use of temporal logic allows the description of program behaviours over time, for example defining future behaviours as the result of past behaviours. Specification languages such as SLIC [17] allow the interactions between software components to be analysed and verified. Some properties exist that are specific to the problem of concurrency: deadlock, where all concurrent processes block on some conditions and none are able to make progress is a universal property; the validity of accesses to shared variables and resources, however, is more difficult to define.

The specification of properties allows verification tools to identify errors in programs, making it possible to examine a particular execution of a program and detect errors. This requires that program execution traces are available against which the properties can be evaluated. At the simplest level, this means that the engineer-written test suites need only provide sample inputs, and the verification process needs to check whether any properties are violated when the software is run against the tests.

A variation on this approach is *runtime verification*, where properties are verified as the program runs, and cause early program termination if a property is violated. While this may appear unwise (a verification technique causing early termination, which is often an incorrect behaviour itself), the detection of a property violation indicates that incorrect behaviour is already occurring, and often early termination is the safest outcome rather than continuing down an incorrect code path [28]. Alternate solutions include entering some kind of fail-safe mode or other emergency handler designed to prevent the detected incorrect behaviour causing any (further) damage.

Fundamentally though, property verification will still suffer the same problems as testing, in that the number of inputs to a program can be huge, and not all operations can be tested in all scenarios. This problem is often referred to as the *state explosion problem*—a *state* meaning a valuation of variables that are used in the program. The solution to this limitation is the development of techniques that do not require repeated execution of the software, and instead utilise some form of abstract interpretation to deduce facts about the program. This does not reduce the number of states the program can have, but instead reasons about them in a much more efficient manner. Such approaches can avoid enumerating the entire state space of the program by approximating the program, or exploiting language properties (such as type systems) to prove states unreachable.

1.3 Automated Verification

When turning to automated verification algorithms we must first familiarise ourselves with their limitations. Firstly, deciding the properties of general purpose program is generally undecidable, for example reachability of a program location. Some algorithms accept this fact and perform verification without any guarantee that an answer will ever be discovered. Others provide an approximate verification, either by *overapproximating* program behaviour by modelling it in a way that might include more behaviours than the program actually exhibits, or by *underapproximating* which covers fewer behaviours than the program really contains. The former can produce *unsound* results, where property violations are reported when the violating behaviour is not actually part of the program; while the latter is *incomplete*, opening the possibility for property violations to exist that are not reported.

1.3.1 Static analysis

One of the simplest and most successful strategies that follows an abstract interpretation approach is *static analysis* [71], typically applied to a high level program representation, often during compilation. Whilst simulating executions through the program, rather than attempting to track the exact state of the program at particular point, an *abstract domain* is used that approximates the set of values a variable may take. For example: the upper and lower bounds of a variable's value may be used instead of the multiple values it could contains during execution. This vastly reduces the number of program states that must be tracked during verification, but does not provide precise information about the execution. Continuing our example, given a property that a variable never has a particular value, we would not be certain that that property is violated if the range of valuation we track covers that value, as we only know the variable's possible values are bounded by that range, not that all values in that range are feasible. However if we had a property claiming that a variable never had a particular value, and our tracking demonstrates a range *not* covering that value, then we can be certain the property is never violated. Another, real-world example of this kind of static analysis is the detection of uninitialised variables, where variable assignments are tracked to discover paths where no assignment occurs before the variable is read: this is often a language property violation.

Static analysis approaches have proven successful at catching simple faults — i.e. property violations that do not depend heavily on program logic or the manipulation of memory. They are particularly effective when integrated into the development process (see below), allowing faults to be identified and solved while code is still being written. Their use for general purpose verification is limited due to the broadness of their approximations, which makes it difficult to be certain that properties depending on complex program logic are violated. They also lack a general method for determining the *cause* of an assertion violation, such as the ability to blame a violation on either a path or program location, due to their approximations [76].

These methods are, however, effective enough to have led to a number of commercially viable static analysis tools being produced, such as Coverity [1] and PVS-Studio [5]. Such tools take a pragmatic approach of integrating into existing development work flows (such as the developers IDE) and highlight possibly defective portions of code. This is a success story where automated verification is helping improve software quality. However, static analysis' limitations are brought into sharp focus by these tools' inability to detect a recent OpenSSL vulnerability [3] commonly referred to as "Heartbleed". The conclusion of both firms regarding their ability to detect the bug is that the logic involved is too difficult for static analysis [49, 108], although Coverity have introduced probabilistic heuristics that now report the error.

1.3.2 Model checking

A more precise verification approach is *model checking*. This was a technique originally developed to verify finite-state models of concurrent systems [56] (specified in tool-specific languages), by systematically simulating each execution path and each possible interleaving of concurrent processes. The process of exploring each possible path in the model allows the property violation search to be sound, and the finite-state limitation allows that search to be complete. Of significant interest is that when a property is violated, a precise path through the model and set of variable assignments is available as a witness that the assertion violation is achievable—a "counter-example" to the correctness of a program. Such a proof is extremely helpful to engineers as part of the fault-fixing process (in some cases allowing *fault localisation* tools to identify the line of code causing the fault).

Model checking has since been extended to address mainstream programming languages. Tracking each state of the program leads to the expected state space explosion, around which techniques such as state compression or state hashing have been developed to improve performance. A more powerful remedy is the use of *symbolic model checking* [119]. Here, exploration of a path through the program stores symbolic representations of the operations taken. The list of operations can then be transformed into a formula, and and be reasoned about algorithmically for a large number of variable assignments, to see whether any lead to property violations. This avoids the necessity to explore every program path with every possible state, instead manipulating a large set of states through one path all at the same time. To address infinite state systems, other techniques define the model checking problem in terms of state transitions, and check a set of paths through a program, rather than attempting to check all states.

Another model checking technique is *predicate abstraction*. For this, the program variables are abstracted into Boolean predicates over those variables, and the program structure is redefined to be in terms of these predicates—a *Boolean program* [18]. Control flow is interpreted as normal, but with the ability to summarise and cache the operation of functions and loops. Condensing the program to a Boolean program reduces the number of states to be explored, but at the cost of accuracy. The abstraction is an overapproximation, and leads to *spurious counterexamples*, a reported property violation that is not actually present in the source program, and is thus caused by the abstraction.

To counter this, predicate abstraction tools often make use of a *counterexample guided abstraction refinement* (CEGAR) loop [52]. After producing a counterexample from a program abstraction, the example is tested against the real program to determine whether it is viable or an artefact of the approximation. If viable, the counterexample is reported to the user. If not viable, the counterexample can be analysed to discover where the current program abstraction is too inaccurate, and it helps determine what portion of the abstraction to *refine* into a more accurate model. This results in a sequence of spurious counterexamples and abstraction refinements, from which the model checker slowly develops an ever more accurate model of the original program. This approach is both sound and complete, but unfortunately non-terminating.

An alternative method to abstract infinite-state programs into a decidable problem is the use of *bounded model checking* (BMC)[37]. Rather than modifying the operation of the program under test, BMC instead bounds the length of program executions that are considered. This immediately leads to an incomplete verification, but does allow for an un-abstracted, precise model of the program under test. The chosen bound on execution length can also be scaled to make the verification fit in whatever memory or time allocation is available, although verification of longer execution runs provides greater assurance of correctness. Other factors can be bounded too, such as the number of times loops have been unwound or the number of heap allocations permitted.

Software model checking techniques tend to condense the logic in the program under test into a form suitable to be evaluated by automated theorem provers. The model checking problem itself is NP-complete [56], and such theorem provers, in the form of SAT or SMT solvers [24], are most suited to solving such problems.

1.4 Design verification

Rather than attempting to verify general purpose software against a series of properties, another alternative is to avoid verifying software entirely, and instead to verify a high level model of the software's operation. This high level model abstracts away fine grained detail of program implementation, and instead must be inspected for a different set of properties, for example, whether a "reset" state is reachable from every path in the model. This approach has been successful when used to verify concurrency properties such as deadlocking. Tools such as SPIN and SMV can check models of concurrent processes that interact through shared variables or message buffers, and verify properties encoded as in-model assertions, temporal logic formulae, and deadlock.

Such dedicated methods of describing the verification task can also lead to greater levels of confidence in correctness. The use of process algebras such as the Psi-Calculus [40] to model concurrent processes allows for formal proofs of correctness to be devised, rather than just checking that no property violation is reachable.

The obvious flaw to this approach is that the models being verified do not necessarily correspond to the actual software that forms the final system. While true, having a formal verified model of how the software *should* operate is a significant step in the development process whereby engineers can know that the system being devised is correct in theory and any fault is an implementation error, rather than a fundamental design flaw.

1.5 Scope of this thesis

This thesis restricts attention to verification of programs in the C programming language. My justification for this is that the most difficult and critical programming environments, such as embedded systems and device drivers tend to be populated by C and C++ software, conforming to additional standards such as MISRA-C [117, p. 1]. Most popular microcontroller architectures are targeted by C and C++ compilers [99, 15, 122, 98], and the programming APIs for drivers in popular operating systems are given in C [123, 109].

Automated verification provides a verification process that applies directly to the source code of a program, without the need for transformation or annotation, allowing code to be verified in situ. The initial barriers to applying automated verification to software are thus low. Within automated verification, model checking provides the most precise technique for identifying errors as it delivers a constructive proof (a counterexample) of such errors. It is also a broad and established field, with 15 different model checking tools taking part in one verification competition [33], representing numerous techniques.

Two of the open questions in the field of model checking are those of property specification and performance. There are multiple ways in which engineers can specify the properties that their systems must respect, typically with limited expressibility, a matter which can be improved. Model checking of software is limited to small programs due to the state explosion problem, and only very small concurrent programs are feasibly verifiable due to even greater state explosion. Therefore, I define my research question in three parts:

1. How can model checking be enhanced to allow more expressive description of properties, specifically through the use of logics such as *linear temporal logic* (LTL).
2. How can the use of automated theorem solvers be optimised for faster verification times.
3. How can software analysis and transformation reduce the amount of time required to verify concurrent programs.

To improve the expressive power of property specification, I explore how the verification of LTL formulae can be performed within bounded model checking. Evaluating LTL properties over the finite traces produced by bounded model checkers is an open and substantial problem; in particular, it has not previously been performed with symbolic software model checkers. I present a technique for symbolically checking LTL formulae over ANSI-C software, and evaluate it on a piece of industrial software and a benchmark suite.

Performance is always an issue with model checkers, as they are subject to the state explosion problem. While such tools face a theoretical limit, the size of software that they can feasibly verify is determined by their performance, which has given rise to events such as the International Competition on Software Verification [31], which seeks to compare the performance (and accuracy) of verification tools. To explore potential optimisations, I examine the way in which model checkers encode program traces into formulae for satisfiability solvers, and how they can be improved.

Concurrent software verification continues to be a unfeasibly difficult challenge. To try and help matters, I study existing verification techniques and try to apply them to bounded model checking. These techniques try to deliver performance improvements through the reduction of the amount of state required to be explored, as well as improving the speed at which each state is explored. In particular I present the results of applying state hashing, an optimal partial order reduction, and using a theorem solver to guide the exploration of state space, to concurrent software benchmarks.

1.6 Thesis structure

In Chapter 2, I explore the fundamental underpinnings of model checking, the various flavours that have developed, its application to both single threaded and concurrent software, and report on some of the leading modern model checking tools and their features. I then declare the features I need to perform my research, and select a model checker (ESBMC) for developing my ideas.

Chapter 3 extends ESBMC to allow for model checking properties written in the temporal logic LTL over bounded program traces.

Chapter 4 studies ESBMCs encoding of program structures to logical formulae, and how different solvers perform when used to solve ESBMCs formulae, and ways in which the formulae can be encoded to enhance performance.

Chapter 5 explores the potential optimisations that are available for improving the performance of multithreaded model checking.

Chapter 6 identifies the current remedial work being done on ESBMCs internal structure, development being performed by other teams working on ESBMC, and the next areas of research I will be investigating.

Appendix A contains the software samples used in the evaluation of the techniques pioneered in Chapter 3.

Appendix B contains result tables cataloguing the performance of tests used to evaluate the optimisations explored in Chapter 5.

Appendix C explains the detail of how ESBMC encodes the model checking problem to a format acceptable for SMT solvers.

Chapter 2

Model checking background

The term *model checking* refers generally to verifying the properties of a formal model, by checking for violations of the properties of that model. There are numerous ways of applying this procedure, over different structures and different levels of expressiveness, with substantially different performance characteristics and trade-offs. Here, I cover the theoretical basis of model checking, its applications, and the limitations under which it operates. I also identify a number of existing model checking tools, discuss their strengths, and the particular software cases they target. The features required for my research are explained, and how the bounded model checker ESBMC [70] fulfils them.

Structure In Section 2.1 I examine the origins of model checking and how it has evolved to support the verification of software. Section 2.2 explores the difficulties found in model checking concurrent software. Section 2.3 studies the use of temporal logics to specify the properties of systems, and how model checkers have been used to verify these properties. Finally, Section 2.4 examines the ESBMC model checker, and explains how it can be used in answering my research question.

2.1 Model checking and Software

2.1.1 Preliminaries

The fundamental idea of model checking [56] is to take the structure of a system and *check* that properties hold over all the states of a system. We will initially explore this idea through finite state machines, which are commonly represented as *labelled transition systems* (LTS) or as *Kripke structures*, of which we shall focus on the latter. To verify this system, we conduct an exhaustive search for property violations, rather than constructing a formal proof of the property truth, which can require significant intuition and creativity from the verification engineer [56].

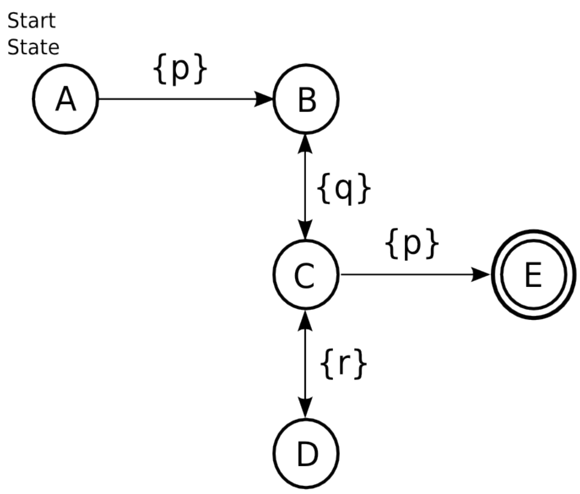

FIGURE 2.1: An example Kripke structure, presented originally in [93, pp. 130]

2.1.1.1 Kripke structure

This form of finite state machine was originally described by Kripke [110], but here I follow the presentation of Holzmann [93] and define Kripke structures as a five-tuple: (S, s_0, AP, R, L):

- S is a set of states;
- s_0 is the initial state $s_0 \in S$;
- L is a set of labels on transitions;
- R is a transition relation defined as $R \subseteq S \times S$; and
- F is a set of final states, $F \subseteq S$.

Figure 2.1 illustrates such a system (with no particular meaning). Note that the semantics of Kripke structures do not require that transitions be bidirectional, or that any particular structure (such as requiring or prohibiting loops) be present.

Here we see the set of states in circles, with names A to E, and the transitions from one state to another as arrows. In this particular structure the transitions are labelled with a set of labels (L) {p, q, r}. A path taken through the structure from the start state is an *accepting run* if it terminates in a state $s \in F$, presented here as a state with two circles, E.

From this structure, we can see that the system can sometimes take multiple paths to the same state in certain places, and that infinitely looping paths can exist. It is also possible for structures to have unreachable states, or distinct subsets of states where there is no path from any state in one set to any state in another set. When considering a system with a large set of states and transitions, reasoning about the structure can become expensive.

2.1.1.2 Properties

In order to check a property over a Kripke structure, we must have an unambiguous definition of the correct behaviour, as described in Section 1.2.

The simplest such property is *reachability*. This asserts that in the structure, there is never a path from the start state to another specified state (often called the "unreachable" or error state). While this may seem to be obvious from the structure of the system, because the transitions are directed, one must actually explore the system to determine the reachable set of states from the start state—which is problematic if the system state space is large. This idea of reachability can be extended to other statements, for example that no state with a particular label attached can be reached from the start state.

The path through a program can also be used to express properties. One might say "after taking a particular transition between states, that the system must then take another specified transition". Extending this, one may make general statements describing the path taken through the Kripke structures involving *time*, with the current state in the system placing constraints upon the future paths through the system, that certain paths must or must not be followed. This form of property is often best understood when expressed in a *temporal logic*.

2.1.2 Model checking of transition systems

In the early 90s, tools were developed that provided a language for describing transition systems and their properties. These tools internally modelled the transition systems as Kripke structures, against which properties were checked. Popular examples of such tools are SPIN [92] and SMV [119]. These were typically used to model hardware systems rather than software algorithms. SPIN uses a description language called PROMELA for state machine description; it is a high level imperative programming language with conditional branching and variable assignment. Its primary purpose however is describing concurrently executing processes, a matter we address in Section 2.2. Properties verified by SPIN can be expressed as reachability conditions by encoding assertions regarding the state of the system at certain points in the model. Temporal properties specified in LTL can be provided and checked, one at a time. SMV specifies its own language for system description, in a form closer to a hardware description language than a programming language, with state machines and their transitions being the primary constructs. SMV also permits properties expressed in CTL to be specified that must hold on the system.

Internally, these model checkers compile the system description language to a Kripke structure with each state representing one valuation of the variables and program counters in the program, and labels corresponding to propositions about the values of variables in the system [93]. The näive approach to checking these systems is to exhaustively explore all paths through the system until one finds a property violation. However, finite state systems with loops can have an infinite number of paths through the system. SPIN records visited states to avoid re-exploring previously visited paths, while SMV computes a fixedpoint over a symbolic formula, explained below.

Another concern of exploring paths through such systems is that the state space grows exponentially as new variables are added to the system—the number of states is the product of all valuations of all variables. This is a significant barrier to model checking any large system, as fully exploring the state space quickly becomes unfeasible. This state explosion problem is a frequent target of optimisations and abstractions to expand the size of program that can be feasible verified. Tackling the state explosion problem is an ongoing battle in the field of verification that has lasted decades and is likely to continue for many years.

Once a property violation has been discovered in a system, the model checker is then able to examine the path that lead to the violation. This path can be formatted into a *counterexample* or *witness*, a constructive proof demonstrating a viable path through the system that leads to a behaviour violating one of the properties being checked. This is one of the most popular features of model checking [55], as it gives the verification engineer a concrete and easily understood proof that identifies the property that is violated, and an example of why this is the case, which can be used to replicate the violation.

2.1.3 Encoding approaches and algorithms

Continuing the focus on model checking transition systems, the available tools differ most significantly in the representation of the current *search state*, the internal state of the algorithm checking the system. The search state can correspond to a particular state in the structure, or perhaps a particular path through it. The first solutions to this problem came from SPIN and similar tools, which are *explicit state* model checkers [93, pp 168]. This is where the state of the system under test is represented as *explicit* valuations of all variables and program counters in the model. These valuations are concatenated in some canonical order to create a vector of bits, corresponding a particular state of the Kripke structure in its entirety. One may then proceed to check the model by the use of a *depth first search*, where each transition through the Kripke structure following from the initial state is explored, storing each distinct state reached into a set of explored states, and backtracking when an already explored state is discovered.

This approach is sound and will terminate, as a finite amount of state space has to be explored. The state space explosion problem, however, can make it unfeasible. The memory required to store all states is proportional to the state space of the system under test; as such complicated models will consume a vast amount of memory. As a result much research has been applied to reducing the memory overhead of such representations: compressing, or even hashing them (discussed in Section 2.2.3.2).

Another way of solving this problem is to formulate a *symbolic state representation*, where the values of variables are not explicitly stored, but their values are *symbolically represented* by the state encoding. One way to rationalise this is to imagine storing the valuation of a variable as a range, instead of enumerating each particular value that it might have in distinct states. Operations on the variable values will require these ranges to be refined and altered during exploration, but the scope for reduction in memory consumption is considerable.

One of the first practical implementations of such an approach was invented by McMillan [119]. Model checking problems are encoding in μ-calculus [46], which describes a fixedpoint: states are labelled as to whether they fulfil the property or not, and a *binary decision diagram* (BDD) [45] is built to evaluate which states can be reached, according to the valuation of variables in the model.

Following the presentation of [45], a BDD is defined as a directed acyclic graph $\langle V, E\rangle$, with V the set of vertexes and E the set of edges. Each vertex has either zero or two children. Leaf vertices are labelled true or false, and all other vertices are labelled with a proposition about variables in the system under test. We consider here *ordered* BDDs [119, p. 41], where every path from the root vertex of the graph has the same sequence of propositions labelling the vertices along the path. A model checking tool (such as SMV) can then summarise the operation of functions (or transitions, or whatever is appropriate) into a decision tree represented by a BDD, with propositions applied to input values, and the true/false labelling of leaf vertex corresponding to whether a program property is violated if the propositions along the path to the vertex hold.

Then, a model checker can repeatedly compose BDDs [46] representing the operations of the system under test (the algorithms for which are not explained here), essentially symbolically exploring the path through the program, manipulating sets of states at a time. When a fixedpoint is reached, all paths have been explored. The produced BDD can then be examined to see whether it contains any leaf nodes signifying a program property is violated—if it does, the path through the program to that violation corresponds to the path through the BDD's propositions.

The pitfall of BDDs is that the memory space savings available depend heavily on the choice of propositions about the variable valuations, and the order in which they are evaluated [119, p. 41]. Some BDDs do not have an optional ordering of propositions at all, with all orderings leading to inefficient amounts of memory usage. In addition, once a BDD has become sizeable, attempting to re-balance it to reduce its size becomes a particularly unwieldy and time consuming task [37]. Finally, the use of a fixedpoint to evaluate what states are reachable prohibits the application of this process to systems that may have infinite state space (see next section).

2.1.4 Bounded model checking of software

Applying the concepts of model checking to software is a much more complex task. General purpose programming languages are Turing-complete, and as such attempting to prove properties such as reachability is *undecidable* [42]—there can be no algorithm that finds whether a particular program state can be reached and always completes [152]. Additionally, general programs can theoretically use an infinite amount of memory. This leaves two options to those wishing to implement a model checker: either accept that their checking software may never complete, or accept some unsound or incomplete approximation that will.

Instead of attempting to transform the program under test into a state machine for evaluation, we can instead apply *bounded model checking* (BMC) to the problem, as described by Biere et al [37]. Program states are still formalised as a valuation of all variables in the program and the location of the program counter, with the set of states potentially infinite. To reason about this, program statements are considered to be transitions between one state and the next, characterised by the transition relation $R(s_i, s_{i+1})$. Starting from some initial state $s_0 \in S$, the program is interpreted as being a sequence of these transitions, each transforming the current program state to a new state. After some number k of transitions has been taken (the *bound*), exploration ceases. The final of state is evaluated to determine whether it violates a property, by checking whether the verification property ϕ holds when using the final state's variable valuation.

Formally, this procedure can be formulated thus:

$$\psi = s_0 \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg\phi_k$$

Here, from the initial state transition function R is applied to the program states k times, to represent the program state after k transitions. The property ϕ is then tested over the final values of the variables in the program. Overall, this formula can only be true if there is a consistent assignment of variables that makes $\neg\phi$ true. If such an assignment does not exist then the property holds; otherwise the property is violated. We denote ψ the *verification condition*, which we test for satisfiability to check whether a property violation exists in the *unrolled* program.

Informally, this formulation of the problem leads to us evaluating a *prefix* of a program execution (assuming the program does not terminate in that time), where the property is checked after the program has made k state transitions. This can simply be a bound on the number of instruction transitions taken, or more elaborate constraints such as bounding the number of times that loops are unrolled. The problem can also be reformulated to check every state in the prefix for a property violation.

Such constraints on the checking of the program under test mean that we now perform an incomplete analysis, referred to as an *underapproximation*. Checking a program up to the bound k allows us to detect that the desired property holds over *some* of the execution; however this does not cover all behaviours, and a property violation occurring after $k + 1$ steps would not be detected. Bounded model checking thus is relegated to discovering the presence of property violations, rather than having the ability to demonstrate their absence. However, if we knew that the program has a maximum length of some n steps, then we could set the bound to that level and still be assured that the model check is complete. Unfortunately determining this value has been shown to be at least as hard in general as model checking [60].

The verification remains sound—the program unwinding up to bound k is still an accurate execution of the program, and as such any counterexample found by the model checker will be a true property violation in the program, as opposed to a spurious error introduced by an abstraction. That being said, some care must be taken to ensure this is the case. If the bound is applied to the number of times that loops are unrolled then we must deal with the case where the program *can* execute a loop further, but the unwind bound prohibits it. In this scenario, the next state of the program is undefined. The simplest solution here is to discard such traces—thus restricting exploration to paths where the program exits the loop before reaching the bound. This preserves soundness, and the discarded traces are victims of the incompleteness of the model check.

This model checking approach also requires a new state encoding approach, suitable to checking a partial trace of the programs computation against a property for consistency, without an explicit definition of a program state. A suitable tool for this is a SAT solver, which takes a propositional formula, explores the possible assignments to the variables in the formula, and finds whether such an assignment can *satisfy* all assertions in the formula, otherwise it is *unsatisfiable*. Recent advances have led to SMT (*satisfiability modulo theory*) solvers, which allow more expressive logics for describing the formula, including first order arithmetic operations and theories of arrays. SMT solvers also allow the use of quantifiers, but at the expense of decidability, thus they are avoided for bounded model checking.

Encoding the bounded trace of the program under test to a SAT solver is straightforward. The state space of the program can be represented as a set of variables in the SAT solver, that are updated with some calculation upon each state transition in the trace. Once the encoding of the formula is complete, assertions are added claiming that a property violation does occur. The SAT solver then attempts to find a valuation of the variables in the program that satisfies all the state transitions and enable the property violation assertion. If the formula is unsatisfiable then no such violation exists, while a satisfying assignment shows that a violation does exist, and the valuation defines the variable values in the program along the violating path.

In addition to supporting the use of bounded model checking, this is also a symbolic encoding (Section 2.1.3) of the model checking problem, similar to BDDs, as it avoids having to explicitly represent explored states in memory. SAT has also been the subject of significant development effort in recent years, with international competitions evaluating the effectiveness of solvers and algorithms. This scenario ensures that SAT based solving is at the leading edge of decision procedure performance. The accepted wisdom is that, while SAT is generally NP-complete, there are numerous subclasses of problems that SAT solvers solve in polynomial time.

2.1.5 Other model checking techniques

We now briefly cover model checking approaches other than BMC. These are methods that perform a model check with some alternate abstraction or approximation. While not the central aim of this thesis, they are of interest in understanding exactly what the benefits of BMC are, as well as potentially inspiring new algorithms for approximating within BMC.

2.1.5.1 Predicate abstraction and CEGAR

The opposite of the underapproximation seen in bounded model checking is to *overapproximate* the program under test, that is, abstract it in such a way that the model being checked exhibits more behaviours than the original program does. If a property holds on this overapproximated system then it holds on the original, but the presence of a property violation does not guarantee that the original system violates that property (the analysis is complete, but unsound). Such property violations are referred to as *spurious counterexamples*.

Overapproximation can still form a useful part of the model checking approach. One such approach is known as *predicate abstraction*: the source program is abstracted from some input language into a *Boolean program*, where all variables are Boolean valued, and represent some predicate on an input program variable. This can then be coupled with *counterexample guided abstraction refinement* (CEGAR): from an initial abstraction of the system, spurious counterexamples are analysed to determine where the abstraction departs most abruptly from the original program. The abstraction is then *refined* to eliminate this path, and the new abstraction is model checked again. When performed iteratively this creates a *refinement loop* where the abstraction of the program under test is slowly refined, until either a counterexample is found that is feasible in the original program or the property is found to hold.

A number of model checking tools make use of CEGAR [18, 54, 35, 36]. One limitation on its technique is that, as it is complete and the property checking problem is undecidable, the CEGAR method is also undecidable. Empirical evidence [18] suggests this does not occur frequently though.

2.1.5.2 K-induction

Inductive proofs are powerful, and would be useful in automated verification. It does not, however, cleanly apply to software or the model checking problem. One could verify a base state of a piece of software, however the inductive step would require proving that if a property holds in any state, it holds in all subsequent states, which is more or less a reformulation of the model checking problem. Sheeran et al. [146], however, in the context of explicit-state model checking of a hardware transition system, determined a method to create an inductive step based on bounded traces of a transition system that was both efficient to produce and gave the necessary assurances of correctness.

For the inductive step, they postulate that it is not possible to reach a violating state after k transitions from any other state. Informally, imagine that we start an exploration in a system at any point, with all variables having free values: proceed k transitions to a state that has a property, and then assume that the property is violated. If this proposition is unsatisfiable, then we know that there is no state in the program that can trigger a property violation after k steps. Sheeran et al. show that for a particular structure of program (loops that have been unrolled k times) this proves that the property will never be violated, if the initial k unrolls of the loop do not trigger a property violation either.

A significant limitation of their algorithm is that, to achieve decidability, they limit the paths explored to only cover those that do not contain duplicated states. This maintains completeness as already explored states need not have their successors checked. However identifying duplicate states is much more difficult to achieve in symbolic model checking. Donaldson et al. [75] and ESBMC [126] have implemented induction for software, however the former requires loop invariants to be provided, and the latter can produce indeterminate "unknown" results.

2.1.5.3 Craig Interpolants

Inductive invariants can be produced by other means. *Craig interpolation* is a method of calculating an *interpolant* between two formulae: a proof of why it is impossible for the two to be both consistent. When considering model checking, for a particular trace we can take the formula representing the variable valuations up to the final state in the trace, and compute an interpolant between that and a formula defining a property violation.

To make use of this interpolant, we can attempt to compute an inductive invariant by taking a proposition, producing a fixed point of states in which a proposition holds, and then show that the interpolant is implied by the invariant. Then, so long as the invariant holds in the initial state too, we can be sure that the property violation state is unreachable, as all reachable states are provably inconsistent with the property violation.

While useful, the use of a fixedpoint means that Craig interpolation cannot be performed on infinite state systems (such as C programs in general), and thus is not of use to us here.

2.2 Model checking concurrent systems

One of the most difficult tasks in engineering is the creation of safe systems that operate concurrently. Examples of this include hardware processes that interact with other hardware processes, and concurrent software processes (commonly referred to nowadays as *multithreaded software*). Crucially, some form of shared resource is required, such as shared memory, queues or message passing interfaces. Otherwise the system merely contains multiple processes running independently of each other. The difficulty arises from the fact that, while a single system can have a

```
void *thread1(void *a){
 while (true) {
  if (x < 5) {
   insert_job_item();
   x++;
  }
 }
}
```

```
void *thread2(void *a){
 while (true) {
  if (x > 0) {
   fetch_job_item();
   process_job();
   x--;
  }
 }
}
```

```
void *thread3(void *a){
 while (true) {
  if (x > 0) {
   x--;
   fetch_job_item();
   discard_job();
  }
 }
}
```

FIGURE 2.2: An example of three threads attempting to synchronise through a shared variable, x. Assume the functions are started simultaneously as separate threads, and that x is initialised to zero.

well understood procedure and set of states, there is (normally) no guarantee about what state another concurrent system is in, when communication occurs. There may also be no guarantee that the progress of time is the same for each concurrent process. This leads to the number of states that the whole system can possess being the product of all the states of all the concurrent processes it contains. This number of states increases exponentially with each new concurrent processes; this is another manifestation of the state space explosion problem. Here, we consider how this problem may be effectively model checked, without committing to a particular model checking method.

Errors that depend on concurrent systems interacting in a particular order, often referred to as *race conditions*, are notoriously difficult to identify and eliminate. Even once it has been identified that there is a concurrency issue, the particular sequence of events happening in a particular order that causes it may happen rarely, and thus difficult to reproduce. For similar reasons, testing for the presence of concurrency bugs is ineffective, as it requires fine grained control of the order in which different parts of the systems execute.

The difficultly of writing concurrent software has been known for much longer than model checking tools have existed [55]. However, the growth in the number of multi-core processors entering the consumer market and the corresponding increase in software using multiple threads of execution has led to a new-found urgency in verifying concurrent software.

The interactions between concurrent systems may also be poorly defined, or even be machine dependant. Take for example, the C program in Listing 2.2, containing three concurrently executing functions communicating through a shared variable x, with one thread generating jobs and the other two consuming them. From the plain meaning of the program, we would expect x to represent the number of unprocessed "jobs" in the program, and for there to be never more than five unprocessed jobs, nor for the consumer threads to call fetch_job_item when there are none available.

Numerous factors, however, can further complicate this scenario. A memory cache could delay writes to the shared variable, access to memory may not be atomic, the processor may re-order writes or the compiler can elect to hold updated values in registers indefinitely. This can lead to the shared variable x appearing to have different values to different threads. These factors can

```c
void *thread1(void *arg) {
  pthread_mutex_lock(&lock);
  progress_signal = 1;
  pthread_mutex_unlock(&lock);
  return NULL;
}
```

```c
void *thread2(void *arg) {
  pthread_mutex_lock(&lock);
  while (progress_signal == 0)
    ;
  progress_signal = 0;
  pthread_mutex_unlock(&lock);
  do_something();
  return NULL;
}
```

FIGURE 2.3: Two threads synchronising using a shared variable, with locking to prevent conflicting access, but a deadlock.

```c
void *thread1(void *arg) {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
  progress_signal = 1;
  pthread_mutex_unlock(&lock2);
  pthread_mutex_unlock(&lock1);
  return NULL;
}
```

```c
void *thread2(void *arg) {
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
  progress_signal = 0;
  pthread_mutex_unlock(&lock1);
  pthread_mutex_unlock(&lock2);
  return NULL;
}
```

FIGURE 2.4: Two threads acquiring locks before updating `progress_signal`, with a fatal lock ordering error

depend entirely upon the architecture of the machine that the code is compiled for, or in even worse cases the actual device that the code executes upon. In the face of this, it is clear that it is necessary for language standards to define how concurrent software behaves and interacts[1], and provide guarantees on what is considered correct behaviour.

Computer scientists have addressed the problems of concurrency by introducing new programming constructs that enable *synchronisation* between concurrent threads. The most common of these is a "lock": one should "hold" a specific lock to access a particular shared resource, and an underlying library or language facility ensures that only one thread may hold that lock at a time (normally by blocking execution until the lock is acquired). This leaves open the possibility that the programmer does not perform the correct locking before accessing the resources, although such conditions can be detected more easily than race conditions. So long as locking principles are obeyed, all accesses to the shared resource are *serialised*, in that threads access the resource only one at a time. Furthermore, run-time and processor optimisation's that might interfere with serialisation are inhibited. This prevents dangerous concurrent accesses, but introduces its own failure mode, that of *deadlock*.

A deadlock is a situation where the entire set of concurrent threads are indefinitely *blocked*, waiting for some condition to become true before continuing to execute. If all these conditions are internal to the process (i.e., will not be triggered by external stimuli), then none of the threads will ever resume execution. Examples of this can be seen in Listing 2.3 and Listing 2.4. In the first, thread 2 acquires a lock before waiting for an action from a second thread to signal it

[1] Such as the recently agreed C11 standard, which specifies system behaviour when memory shared between threads is accessed.

should continue execution, while thread one attempts to acquire that lock too before signalling. In the second example, we see a *lock ordering* error—two threads attempting to acquire two locks in different orders, which can result in each thread having one lock, and neither being able to continue until the other lock is released.

2.2.1 Verification approach

The verification of these systems is difficult—how can we represent concurrent actions in a state transition system? An obvious approach is one of *sequentialisation* [115], where one formulates a single threaded program that accurately models one behaviour of the multithreaded program. One way to achieve this sequentialisation is by *interleaving* the concurrent threads of a program. This is where, for a set of threads, one nondeterministically selects a thread to be executed and performs the next transition available to that thread. The product of this approach is a sequence of transitions from different threads, representing a single threaded trace of steps, against which one can then attempt to apply normal model checking techniques. The success of this approach depends on the system not having any truly concurrent behaviours [115]: a particular piece of memory for example can only ever be written or read by one thread at a time, and apparently concurrent accesses to it eventually becomes sequential accesses in some order. Should a resource actually support more than one operation on itself in a single step, and the result is not representable by interleaved single steps, then this approach fails.

Consideration must also be given to how threads are *scheduled*, i.e. which of the available set of runnable threads will be run, and in what order. An acceptable schedule is to run one thread to completion followed by running another thread to completion, and so forth. This assumes that all threads either terminate or have their execution bounded in some way. That schedule may be a valid behaviour of the program, but not particularly useful, as it does not check any complex interaction between threads. To fully explore and check all behaviours that the program can exhibit, one must thus explore all schedules of the concurrent threads, to cover all the states in the system that can be reached through interleaving. This is the realisation of the state explosion problem, in that the state space to be checked increases exponentially as we introduce more states to individual threads.

Switching which thread is being executed while creating the trace is called a *context switch*, in reference to the way in which processors change what thread they are running.

The process of exploring all the schedulable traces is known as computing the *reachability tree*, which represents all the global states reachable through interleaving threads. Each edge of the tree represents the execution of a transition, each vertex represents a state in the program where a context switch may be taken, and has one child edge for each thread transition that may be taken in that state. The path from the root node to a leaf represents the entire execution path of a single trace with all threads run to completion.

Some effort has already been put into identifying concurrency errors without having to resort to formal verification. LTL properties have been verified in the course of runtime verification for example [28]. Other tools capture thread executions during test procedures, and attempt to replay them with threads executing in different orders to reach unexpected states—CHESS [131] follow this approach. These methods do not allow for discovery of faults ahead of time however, and so will not be examined further.

2.2.2 Encodings for multithreaded verification

The explicit model checking method of SPIN and similar tools extends naturally to support explicit sequentialisation. The depth first search of reachable states can be augmented with the state vector containing the global state and the state local to each thread, then allowing the search algorithm to execute a transition from any available runnable thread. The search then covers all states reachable as the result of any possible thread interleaving.

This depth first search technique can be applied to BMC too. ESBMC's primary search method [68] is to unroll each thread to completion while interleaving thread transitions (in an initial arbitrary order), and store each state along the way. Then, when the current exploration path has no further transitions and if a property violation has not been found, it backtracks to the previous state, and then takes a different context switch. Repeatedly applying this method, and backtracking further when all context switches from a particular state have been explored, ensures that all possible interleavings are considered. In effect, the entire reachability tree is being exhaustively explored from an initially deep path, and working backwards from there. This is essentially the same depth first search algorithm as SPIN, but without the storage of explored states (which does not affect decidability as it only operates on bounded traces).

More symbolic approaches are possible too. Another search method of ESBMC is to perform the same depth first search as describe above, but instead of checking each trace produced by the scheduling algorithm individually, a single SAT formula is composed out of all the traces explored. This follows the same depth first exploration path as described above, but rather than checking for property violations along each explored path, the formula for each path is accumulated into the same SAT formula. The SAT test of the resulting formula then checks all paths down all interleavings of the program. This approach attempts to trade additional formula size for reduced solving time: the single SAT formula containing the entire reachability tree is larger than the single path formulae from the depth first search approach, but the solver is able to eliminate entire subtrees of paths if it determines a particular transition cannot be taken. Empirical evidence [68] suggests however that this approach is inferior to the depth first search due to memory usage as well as SAT solving time.

Much more radical symbolic encodings exist, with several tools leaving the exploration of interleavings to the SAT solver. The Lazy-CSeq tool [100] transforms concurrent programs to partially execute portions of threads and, using general BMC tools, allows nondeterminism in the

```
void *t1(void *arg) {
  int *ap = arg;
  int local_id = 1;
  ap[x++] = local_id;
  return NULL;
}
```

```
void *t2(void *arg) {
  int *ap = arg;
  int local_id = 2;
  ap[x++] = local_id;
  return NULL;
}
```

```
void *t3(void *arg) {
  int *ap = arg;
  int local_id = 3;
  ap[x++] = local_id;
  return NULL;
}
```

FIGURE 2.5: Three threads synchronising an index through the `x` variable, and a mixture of local and global operations. Assume that all three functions are started as threads simultaneously, that x is initialised to zero, and the argument to each thread is an array of at least three elements in size.

program to arrange the different orders in which portions can be arranged. CBMC [11] executes threads to (bounded) completion, replacing accesses to shared memory with special variables, and allows the SAT solver to arrange the order in which shared memory accesses occur between threads. Both these techniques are examples of encoding the exploration of interleavings into a single formula where the solver can control the order of interleavings.

2.2.3 Optimisation

The most significant barrier against the application of model checking techniques to real-world concurrent systems is the state explosion problem. Thus our attention turns to how one can check properties of such systems while keeping the number of states that one has to explore small, a matter that is the topic of much research in the field. This subsection explores existing approaches, with my own contributions discussed in Chapter 5.

2.2.3.1 Partial order reduction

One observation made regarding the state explosion problem when checking concurrent software is that, while a large number of thread interleavings exist in relatively small programs with a small number of threads, the set of distinct states arising from these interleavings tends to be smaller. Take for example Figure 2.5, in which three threads store an integer into an array, at an index determined by the global variable `x`. Assume for this example that we execute each statement of the program atomically and interleave at the end of each statement. Each thread has several states, and we could conceivably interleave the statements in 64 different orders. However, there are still only six different global states that the program may end in once each thread has completed.

To understand why this is, consider the concurrent system as a Kripke structure, where each state represents the global system state and all instructions in all threads form the transitions between states. In this system, we can classify a pair of transitions (in different threads) as either being *independent* or *dependant* of each other. Intuitively, this indicates whether or not the pair, if executed one after an other, will result in different states depending on the order. A *partial order reduction* (POR) is where we identify independent transitions and only explore one ordering of them: all system states are still explored, but with fewer interleavings.

This approach is formally presented in [86], which we follow here. First, define T as the set of all transitions possible in the Kripke structure. Transitions are said to be *enabled* in a particular state if that transition can be taken. We represent a transition $t_1 \in T$ as $s \overset{t_1}{\rightarrow} s'$, where $s, s' \in S$ are the source and destination state of the transition, respectively. Let $D \subseteq T \times T$ be a relation between a pair of transitions. If the pair $(t_1, t_2) \notin D$, this signifies that the two transitions are independent, and throughout the state machine this means that:

1. If t_1 is enabled in state s, with target state s', then either t_2 is enabled in both s and s', or in neither. Thus, t_1 cannot disable or enable the t_2 transition. The inverse must hold too, i.e. t_2 may not enable or disable t_1.

2. If both transitions are enabled in s, then both $s \overset{t_1 t_2}{\rightarrow} s'$ and $s \overset{t_2 t_1}{\rightarrow} s'$, i.e. if both transitions are taken, but in a different order, then we still reach the same state.

Applying this to the state space exploration of a concurrent system, this means that if either of two transitions t_1, t_2 may be taken, and $(t_1, t_2) \notin D$ then we may pick one ordering of the two transitions and not explore the other, and be assured that we are still exploring all reachable states. Only dependant transitions need different orders exploring.

The difficult task is identifying the dependency relations between transitions, particularly when factors such as pointer indirection arise which makes it difficult to determine which variables are read and written in a particular expression. Approximations of the classification that do not reduce the completeness of the search can be useful, however.

In the context of explicit-state model checking, initially partial order reductions computed different sets of transitions from a particular state, for example persistent-sets and sleep-sets [84], that identified which transitions must be explored and which need not, respectively. These techniques statically analyse the structure of the model, and cannot be effectively used to verify general software where features such as indirection cannot be statically analysed [81].

Instead, [81] suggests a *dynamic* partial order reduction, analysing software as it is explored and identifying dependant interactions, in the context of *stateless search* model checking [103]. Here, a scheduler explores thread schedules instead of states, an approach used for example by the CHESS [131] and Verisoft [85] model checkers, and uses the partial order to identify where the schedule should backtrack to during exploration. Research into the most efficient method for this continues [9]. These techniques constitute an explicit exploration of interleavings, but in a nonterminating manner: they do not store states or detect cycles in the state space, and so never terminate or have a guarantee of completeness.

Other techniques support the symbolic encoding of partial order constraints, as explored in [159, 106]—however, they only ever deal with finite state abstractions of the program under test.

Applying partial orders to model checking (that fully explores state space) of concurrent general programs is uncommon [11] and has few examples [11, 66, pp. 122]. We shall revisit partial order reductions for concurrent software verification in Chapter 5.

2.2.3.2 State hashing

Given that we wish to eliminate redundant states that do not need to be explored, we may just generally say that we wish to avoid exploring duplicate states. This approach is already taken by the depth first search of model checkers such as SPIN. The memory consumption for storing these states and time spent searching for duplicates, however, increases as one might expect from state explosion. A method developed to tackle this is to, instead of storing the explored states themselves, compute and store a *digest* or *hash* of that state, which uses much less memory when stored [95]. One can then detect the exploration of duplicate states by computing the hash of the current state, and looking up whether that hash has been explored before.

A hash can be thought of as a summary of some information: the input state data is passed through a mathematical transformation to become another piece of data (the hash) of fixed width, typically only a few bytes. This item can then be stored in an ordered set or some other structure that can test for the existence of an item quickly. SPIN takes a different approach [93, pp. 206-209] and instead allocates an array of bits (of size 2^n, where the hash value is of bit width n), marking the bit corresponding to the hash of an explored state as 1 and unexplored states 0.

Hashes are designed to make it unlikely for different inputs to produce the same output; they are, however, fundamentally an abstraction of the input that loses data, meaning that it is possible (however unlikely) that when processing a previously unexplored state, the hash value of its state vector will match the hash value of a different, already explored state. This is referred to as a *hash collision* [95]; the effect is that an unexplored state is discarded due to the incorrect appearance that it has already been explored. This damages the completeness of the model check; checks performed using state hashing will still generate sound counterexamples if a fault is found, but cannot be used to guarantee the absence of faults.

State hashing has proved popular in explicit state model checking, appearing in SPIN [92] and other popular model checkers [39, 21, 129] applied to system models and real-world software alike. State hashing has not been applied to symbolic model checking techniques, as they manipulate sets of states in each operations.

2.2.3.3 Context bounding

Once we concede completeness in multithreaded model checking, *context bounding* becomes a viable option. The number of context switches taken between threads during scheduling is bounded — simply implemented by inhibiting any further switches between threads after the bound has been reached. The effect is that the reachability explores all paths that contain that number of context switches, and none that exceed the bound, damaging completeness. The reduction in number of interleavings that must be explored is vast: whereas before n^t interleavings would be explored (for n Instructions in the trace, with t threads), we instead explore $n \times k^t$, with k the context bound. This bounded analysis is still useful as empirical evidence indicates that many multithreaded program bugs can be discovered in a relatively low number of context switches [138, 130].

2.2.3.4 Symmetry reduction

Another technique for state reduction is reducing *symmetry* through identifying equivalent subtrees of states and relations in the model, and subsuming them into a reduced subtree [57]. This amounts to computing *orbit relations* that reduce the original model to a quotient model, which has been shown to be NP complete [57], and is a serious barrier to reducing symmetry [124] in symbolic state representations [79]. Explicit state model checkers have found efficient applications of symmetry reduction, however [91, 41, 39]. A detailed study of symmetry reduction techniques can be found in [74].

2.3 Temporal logic

Temporal logics take propositional logic and extend it with modal operators related to time—expressions can be made about the *past* and *future*. These logics are important in system verification, as they allow a richer definition of the correct behaviour of the system, for example that "whenever a request is received, a response is delivered", which fundamentally requires that past states affect the validity of future states. While it is possible to jerry-rig the system under test to store facts about past events and test them during verification, thus expressing them as reachability, it is invasive and much more difficult to understand.

We say that a temporal formula *holds* over a system if all possible executions of the system satisfy the behaviour described by the formula. If any execution does not comply, then the formula is *violated*.

The most popular temporal logics are *linear temporal logic* [136] (LTL) and *computation tree logic* [56] (CTL). These express properties about the system under test in terms of *traces of states* and *paths*, respectively, also referred to as *linear time* and *branching time*. The difference is that expressions about the future in LTL refer to all the future states that the program may reach, whereas CTL quantifies over future paths that the program may take: the result is that some properties can be described in CTL but not in LTL, and vice versa [55].

To briefly illustrate the difference: LTL cannot express any CTL formula that requires an existential quantifier, as LTL does not have any explicit quantifiers itself (it is implicitly quantified over all traces of states). The LTL formula $\mathsf{FG}p$ ("In the future p always holds") can be expressed in CTL as $\mathsf{AFAG}p$, however it will not hold over systems where an infinite number of paths can be generated that pass through a state $\neg p$, as not all *paths* have a future where $\mathsf{AG}p$ holds, regardless of whether such a state is eventually reachable.

Empirical evidence suggests that engineers find LTL formula easier to understand compared to CTL [29], as it requires thinking about all computations simultaneously, as opposed to single computations individually. For this reason, LTL is more commonly used in model checking of software [38, 97, 104], and as a result I will focus on LTL for the rest of this thesis. The combination of all the CTL and LTL operators, however, make up the more expressive logic CTL*, which can express all behaviours from both sublogics, as well as some behaviours that can be expressed by neither.

2.3.1 Formulation

In the standard semantics [136], LTL formulae are interpreted over *traces* over a given alphabet Σ of symbols, i.e., possibly infinite words $a_0 a_1 \cdots$, with $a_i \in \Sigma$. In LTL model checking, it is common to consider a non-empty set of atomic or primitive propositions Prop and to define $\Sigma = 2^{\mathit{Prop}}$. In the context of our work, each symbol $a \in \Sigma$ denotes a valuation, the set of Boolean expressions over the global variables of the C program that hold at a given time; it can be seen as a possible world in a Kripke structure. We use $u \in \Sigma^*$ to denote finite traces, $w \in \Sigma^\omega$ to denote infinite traces, and ϵ to denote the empty trace. We further use $w^i = w_i w_{i+1} \ldots$ to denote the suffix of an infinite trace; for a finite trace of length n, $u^i = u_i u_{i+1} \cdots u_{n-1}$ if $i < n$ and ϵ otherwise. We finally use the notation a^ω to denote the infinite trace consisting of the letter $a \in \Sigma$ only.

Following our presentation in [128], we define the operators of LTL thus:

Definition 2.1. LTL formulae are defined over primitive propositions, logical operators and temporal operators as follows:

$$\begin{array}{rcl} \varphi, \psi & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{p} \mid \neg\varphi \mid \varphi \vee \psi \\ & \mid & \mathsf{X}\varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \varphi \,\mathsf{U}\, \psi \mid \varphi \,\mathsf{R}\, \psi \end{array}$$

Here, p is some proposition about the state of the system, for example whether the current state in a Kripke structure has a particular label. The Boolean constants and logical operators $\neg$ and $\vee$ are defined in the usual way, with $\wedge$ and $\implies$ following. The temporal operators are "in the next state" or *next* (X), "in some future state" or *eventually* (F), "in all future states" or *globally*

(G), *until* (U), and *release* (R). $\varphi \, \mathsf{U} \, \psi$ means that φ must hold continuously until ψ holds; ψ must eventually become true. $\varphi \, \mathsf{R} \, \psi$ means that ψ must hold now and continue to hold either until φ becomes true as well, or forever (if φ never becomes true). All temporal operators can be defined in terms of X and U [119], but the fuller logic makes LTL easier to understand.

While LTL does not feature any quantifiers, all LTL expressions are implicitly quantified over all the traces that a system may produce. This limits LTL to statements that must always, or never, hold.

Some simple examples of LTL formulae are as follows: to say that a condition p must never be true, we would write $\mathsf{G}\neg p$. A requirement that if a condition p is ever true (for example: "a request is made") that the condition q eventually becomes true ("a response is given"), we would write $\mathsf{G}(p \implies \mathsf{F}q)$. To say that every request received a response, we could write $\mathsf{G}(p \implies (p\mathsf{U}(\neg p\mathsf{U}q)))$, which holds if p is always eventually followed by q, without p becoming true in the meantime.

2.3.2 Büchi Automata

It has been shown [155] that LTL formulae can be exactly converted into *Büchi automata* which accept the same set of infinite traces that the LTL formulae hold over. Evaluating LTL formulae as an automata is a staple of model checking techniques, as we shall see in the next section.

Büchi automata (BA) are finite-state automata over infinite words first described by Büchi [47]. We follow Holzmann's presentation [93] and define a BA as a tuple $B = (S, s_0, L, T, F)$ where S is a finite set of states, $s_0 \in S$ the initial state of the BA, L a finite set of labels, $T \subseteq (S \times L \times S)$ a set of state transitions and $F \subseteq S$ a set of final states. B may be deterministic or non-deterministic. A *run* is a sequence of state transitions taken by B as it operates over some input. A run is accepted if B passes through an accepting state $s \in F$ infinitely often along the run.

2.3.3 Existing verification techniques

Verification for properties expressed in CTL tend to represent the state space of the system symbolically [46, 119, 120], computing a fixedpoint of all reachable paths to find any that violate the given property. As mentioned, we will not be studying CTL in this thesis.

The accepted method of verifying a LTL formula over a system is to interpret the propositions of the formula as propositions over the systems variables, convert the formula into a BA, and then exhaustively explore all paths through the program with the BA consuming each state as an input. Over an infinite trace of states, the BA accepts if the formula holds.

In practice, as infinite traces of states cannot be fully computed, tools instead search for accepting loops through the system under test [162]. SPIN [92] inverts the LTL formula and then performs a depth first search for loops through the system under test where the BA will accept—if one is found, then the inverted formula holds, and thus the original property does not. This inverted formula is frequently referred to as the *never claim*.

Explicit exploration of state space for LTL model checking has been shown to be PSPACE complete [147], and in [58] Clarke et al. show that LTL model checking can be transformed to a form verifiable by symbolic CTL model checkers. Sebastiani et al. [145] study several different encodinigs and propose their own, while Rozier et al. [144] survey all the available LTL model checking tools and find the symbolic approach to be substantially better than any other. Biere et al. [37] show that LTL can be model checked through a bounded trace of state transitions, but only if the finite bound could be determined. A full survey of symbolic model checking techniques is in [143].

Of all the model checking techniques I examined, none were directly applicable to the verification of software, instead limiting themselves to either models extracted from software [94, 80] or embedding fragments of C in the verified model [151, 93, pp. 495]. These approaches limit the verification model to a finite state space, thus avoiding the underlying problem, which is that the infinite traces LTL is defined over cannot be evaluated over models with infinite state space. Bounding the length of the program explored [53] allows property violations within the explored prefix to be identified, but is only of use for finding violations of safety properties.

A verification technique related to model checking is *runtime verification*, where a monitor within a system continuously observes its operation, identifying whether the system is violating a property. Such properties can be specified in LTL [16] and verified on-the-fly, however with finite traces it cannot always conclude that the property holds or is violated, instead yielding an indeterminate result [26]. Bauer et al. study different semantic interpretations of LTL formulae with regard to such finite traces, and propose their own to give meaning to indeterminate results [27].

We shall revisit LTL model checking in Chapter 3, with particular focus on how to evaluate LTL over C programs and how to interpret LTL over finite traces.

2.4 ESBMC

Rather than developing a new model checker in the course of my research, I instead continue the development of ESBMC [70] which is based at the University of Southampton. ESBMC is already a reasonably mature ANSI-C model checker, and none of the aims of my research require an entirely new model checking technique, therefore a large amount of development can be avoided.

In this section I examine the features that ESBMC possesses, the technique used to verify single threaded programs, and its techniques for verifying concurrent software.

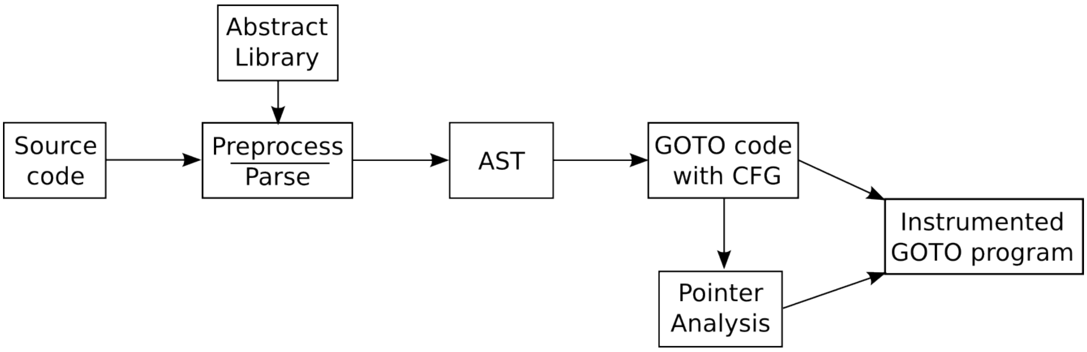

FIGURE 2.6: ESBMC's compilation procedure

2.4.1 Features

ESBMC is a fork of CBMC[2], a bounded model checker with support for verification of in-program assertions and language safety properties. Loop unroll lengths can be bounded, and the problem is formatted into a SAT formula that can then be solved. Property violations result in a satisfiable model in the SAT solver, which can be extracted and formatted into a program path and set of variable assignments demonstrating a counterexample.

ESBMC was initially developed to extend CBMC to support producing an output formula in SMT format [70], suitable to be solved by a SMT solver. This is a more expressive logic, allowing all first order operations, with many theories covering bitvector arithmetic, integer arithmetic, arrays and more.

Further effort has gone into allowing the verification of concurrent threads, modelled using the `pthread` API for POSIX systems [68], with several techniques for exploring multithreaded state space evaulated.

Future development of ESBMC includes implementing support for K-induction [126], and enhanced support for model checking C++ and its libraries [140].

2.4.2 Single threaded verification method

An overview of ESBMCs verification technique is presented in figures 2.6 and 2.7, and is similar to CBMCs. The initial steps, shown in Figure 2.6, are to parse the C program to be tested and instrument it. Parsing C necessitates a run of the C preprocessor, to include appropriate headers and translate lexical `#defines`. This is performed by a built in copy of the Portable C Compiler (PCC)'s preprocessor, which enables ESBMC to intercept the inclusion of certain system headers and replace them with its own—necessary to ensure that architecture specific features of the host machine are not compiled into the program under test. The preprocessed output is then parsed into an abstract syntax tree in the usual manner, then translated into the

[2]Version 2.9

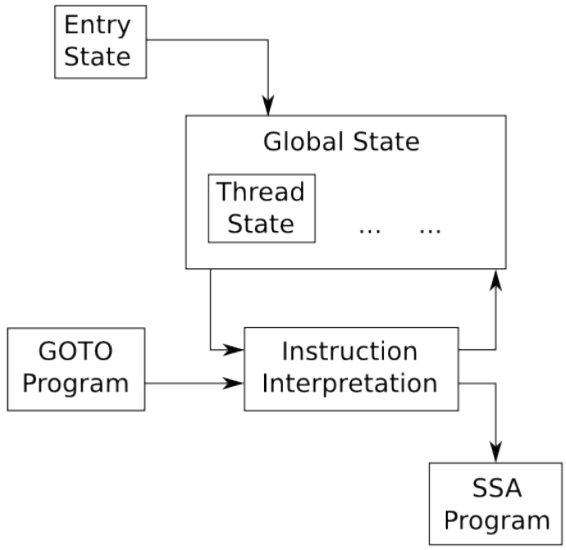

FIGURE 2.7: ESBMC's symbolic execution procedure

internal instruction representation: GOTO code. Models of library functions that have been precompiled to GOTO are linked in before the GOTO code is interpreted.

The GOTO language is one of variable assignments, branches, and very little else. All alteration of data is converted into an assignment statement, and all control flow facilities are flattened into (possibly conditional) branches. A few higher constructs such as function calls are kept. Expressions on the right hand side of an assignment are permitted to be any side-effect and function call free C expression. This representation of the program separates two concerns: the first being the exploration of the control flow graph, the second being the calculation of variable values through the program.

The control flow of C functions is nontrivial to explore, as it contains numerous constructs for loops and conditional execution, while also providing keywords such as `continue`, `break` and `goto` to allow the programmer to break out of such constructs. To avoid having to interpret these constructs during model checking (and the numerous ways in which they can be nested), they are flattened to branches between basic blocks of code. This allows the exploration to be very simple, following the path of branches, with the only complexity being the backwards jumps that loops turn into (the exploration of which must be bounded) and conditional branches.

Once this GOTO code representation is built, a pointer analysis is applied to the program. A fixedpoint is computed identifying the data objects that each pointer variable may point to, at any time during execution. The resulting point-to set is then used to synthesise assertions about the validity of pointer dereferences at runtime (i.e., that the pointer is within the bounds of each object it may point to).

Symbolic execution then proceeds to interpret each GOTO instruction, unrolling loops, and produce a program consisting of *single static assignment* (SSA) instructions. Such instructions assign to each variable in the program once, and it remains immutable. This is illustrated in Figure. 2.7—from some start state, we progressively take the current program state and the current instruction, compute the local and global state changes it causes, and record those changes while emitting an SSA instruction. The SSA program contains the logic of the program, while the state tracking holds information needed to produce future SSA instructions. This includes the set of variables a pointer currently points at, the function call stack and instruction pointer of each thread, and the results of constant propagation of certain assignments.

The number of times a loop has been unrolled is tracked, as is the recursion depth (which is also bounded). Joins in the control flow graph are represented as SSA *phi* instructions, where the values of variables are assigned according to which control flow path was taken through the program.

At the end of (bounded) execution, the SSA program contains assignments representing the value of every variable in the program, at every point in time of the execution. To check for property violations, ESBMC converts the SSA program to a quantifier free first order logic representation: each variable becomes an SMT variable, and assignments are made by asserting an equality between the variable and its formula. Two logics are supported: `QF_AUFLIRA`, a logic over integers, reals, arrays and uninterpreted functions; and `QF_AUFBV` which works over bitvectors, arrays and uninterpreted functions. The different logics have certain tradeoffs—`QF_AUFLIRA` tends to be faster, but cannot precisely replicate C semantics, such as integer overflows and byte representations of all data objects. A more complete presentation of this process is contained in Appendix C.

After the SMT formula is solved by a solver, either the satisfying assignment is used to print a counterexample showing a property violation in the program, or if the formula unsatisfiable, then there is no violation and ESBMC reports a successful verification.

2.4.3 Multithreaded verification technique

In contrast with other concurrent software model checkers, ESBMC uses a mixture of symbolic and explicit state exploration to verify multithreaded software. Variables in the program under test are encoded as described above, with a symbolic SSA program and SMT formula, however the interleavings of threads are explored explicitly.

ESBMC implements this by creating a set of threads, executing each as a sequential program with operations stored (as SSA assignments) to a global record, and context switching between the set of active threads when access to a global variable is detected. There are three variants of this technique, of which we only consider the *lazy* approach, as evaluation [68] has shown the other two to be less efficient. For the lazy approach, ESBMC explores one interleaving of the program to completion, tests the SMT formula for satisfiabiliy, and if no property violation is found then ESBMC backtracks along the path taken and explores a different interleaving.

Interleavings are explored by following the execution of one thread until it accesses state shared between threads (such as global variables), executes a *yield* intrinsic, or creates / joins another thread. At this point in exploration ESBMC stores the current state for backtracking, then explores the path taken if control were to context switch to any other thread in the program. Once a path has been fully explored, and all possible context switches from the current state have been explored, ESBMC backtracks to earlier states to explore other interleavings, or if there are no further backtracking points reports a successful verification.

2.5 Summary

We have studied a series of approaches to performing software verification through model checking, the method of specifying correctness properties of software, and how the resulting model is checked for property violations. A number of modern model checkers are examined for their suitability, and ESBMC is picked for development as it possesses the required features and has the greatest amount of support available. Finally, we have looked in more detail at ESBMCs verification approach, from a high level.

Chapter 3

Checking LTL properties against bounded traces

Model checking has been used successfully to verify actual software (as opposed to abstract system designs) [158, 34, 59, 22, 61], including multi-threaded applications written in low-level languages such as ANSI-C [68, 139, 113]. This approach is typically used for the verification of safety properties expressed as assertions in the code, but it can also be used to verify properties such as the absence of global or local deadlock.

Many important requirements on the software behaviour can, however, be expressed more naturally as liveness properties in a temporal logic, for example we may say about a battery charging device "whenever the start button is pressed the charge eventually exceeds a minimum level". Such requirements are difficult to check directly as safety properties; it is typically necessary to add additional executable code to the program under test to retain the past state information. This amounts to the *ad hoc* introduction of a hand-coded state machine capturing (past-time) temporal formulae.

In this chapter, we explore methods for checking properties expressed in future time *linear temporal logic* (LTL), as defined in Section 2.3, against software written in the ANSI-C programming language, using ESBMC. The work covered here has been published in several papers [125, 128, 127], the presentation of which has been co-authored with my supervisors Denis Nicole and Bernd Fischer, as well as my colleague Lucas Cordeiro. In particular, the presentation in Section 3.2.1 was produced largely with the help Prof. Fischer, and the analysis of Section 3.2.5 was done with a large contribution from Dr Nicole.

We use context-bounded model checking to validate single and multi-threaded C programs against LTL formulae over expressions in the global variables of the C program under test. Thus, if the C variables `pressed`, `charge`, and `min` represent the state of the button, and the current and minimum charge levels respectively, then we can capture the requirement above with the LTL formula $\mathsf{G}(\{\texttt{pressed}\} \implies \mathsf{F}\{\texttt{charge>min}\})$.[1] We check these formulae following the usual approach to LTL model checking [62, 93]; we convert the negated LTL formula (the so-called *never claim* [92]) into a Büchi automaton (BA, Section 2.3.2), which is composed with the program under test. If the composed system admits an accepting run, the program violates the specified requirement.

Our approach differs from previous techniques in two key aspects. First, we check the actual C program, rather than an extracted and abstracted model. We thus convert the LTL formula's BA further into a separate C *monitor thread* and check the interleavings between this monitor and the program using ESBMC. We bound the execution of the monitor thread in such a way that it still searches for accepting loops after the program has reached its own bound. We thus consider the bounded program as the finite prefix of an infinite trace where state changes are limited to this finite prefix; this gives us a method to uniformly check both safety and liveness within the framework of bounded model checking.

Our approach avoids any imprecision due to translating the C program into a BA, but the monitor has to capture transient behaviour internal to the program under test. The monitor and the program communicate via auxiliary variables reporting the truth values of the LTL formula's embedded expressions. Our tool automatically inserts and maintains these on-the-fly and also uses them to guide ESBMC's thread exploration.

The work here describes the first mechanism, to the best of our knowledge, to verify LTL properties against an unmodified C code base, and against multithreaded programs that use the standard pthread library [102]. It is also the first work to create a symbolic LTL model checker that does not use binary decision diagrams (BDDs), through the use of ESBMC. Finally, we also extend the truth domain of LTL properties to give more meaningful information about the liveness and safety properties of potentially non-terminating programs.

Organisation In Section 3.1 we cover the work in [125], describing the conversion of LTL formulae into a form that can be applied to an unmodified ANSI-C code base, and perform some initial experiments with this process.

Section 3.2 addresses some of the performance concerns encountered in Section 3.1, and extends the truth domain of the LTL formulae to provide more fine grained information about certain properties. We apply the process to the same experiments and a case study.

Finally, in Section 3.3 we exercise the process described here with the 2012 RERS Grey-box Challenge [96], a test suite of reachability and LTL properties, to demonstrate the effectiveness of our approach.

[1] Here and throughout the chapter we enclose the embedded C expressions in curly brackets and typeset them in fixed width font.

3.1 Checking LTL properties against ANSI-C software

3.1.1 Linear-time Temporal logic

LTL is a commonly used specification logic in model checking [38, 97, 104], and was introduced in Section 2.3. To express properties about the behaviour of software, rather than only logical propositions, we allow propositions in the logic to be side-effect-free Boolean C expressions over the global[2] variables of the C program.

We interpret a possibly multi-threaded C program as a Kripke structure whose state transitions are derived from the possibly interleaved execution sequence of C statements and whose valuations are the possible values of the program's global variables. We use a separate run of ESBMC to assure deadlock freedom. We finally describe the desired liveness property ϕ as an LTL expression in the above syntax and then check that there are no possible infinite sequences of program states for which ! ϕ holds.

Checking C software with LTL properties and a bounded model checker faces several problems, foremost being that a *bounded* model checker evaluates only finite traces of the program under test (whether single or multi-threaded), and LTL is defined over infinite traces. Below I outline our solution to this problem, how C is interpreted as a trace, and how LTL can be evaluated over such traces through the use of Büchi Automata (BA, see Section 2.3.2) and monitors.

3.1.1.1 Finite traces

The finite traces that ESBMC produces are created by bounding the number of times that loops are unrolled. If the program contains at most one loop that has its unrolling bounded then the finite traces are all prefixes of the potentially infinite traces of the original program. If the program contains several such loops then we can still analyse it, using the `--partial-loops` option. In this case, however, the observed finite traces are not necessarily proper prefixes of the original program traces, and our approach can produce false results, as the symbolic execution can continue past unsatisfied loop termination conditions.

To check LTL against these traces, we chose to extend the finite traces traces *stutter extension* [93, pp. 130] or *infinite extension* [25]. To stutter extend, we take some finite trace of states u^i generated by ESBMC, take the final state of the trace $a_i \in \Sigma$ and repeat it infinitely as the trace a^ω. We then append it to the finite trace, making the infinite trace $u^i a_i^\omega$. In terms of the state of the C program, this would mean that the final state of the program (i.e. the valuations of all variables) is repeated forever after ESBMC's exploration bound is reached.

[2]We consider all global variables to be volatile, in that they are never optimised out of the program, and their modifications become visible immediately

3.1.1.2 Trace semantics for ANSI-C

ESBMC compiles the C source of the program under test into a language of assignments and guarded branches, as explained in Section 2.4.2. Each assignment creates a new valuation of the variables in the program, corresponding to one state in the LTL trace. In Lamport's [115] definition of LTL, the variable valuations correspond to states and the assignments to actions. These assignments, however, bear only a weak relation with the statements and sequence points of the C language, as the order in which side-effects and expression evaluation occur is left undefined by C. For this reason, we do not provide a definition for the LTL X operator, as the next state in the LTL trace does not precisely correspond to any C language construct.

We follow Lamport's [115] definition of LTL, and are only interested in temporal formulae which are closed under stuttering. Our LTL expressions are thus insensitive to refinements of the timestep to intervals less than those required to capture the ordering of changes in the global state. The timesteps only need to be sufficiently fine to resolve any changes in the propositions that the LTL formulae are expressed over, so we only register a timestep when any global variable that an LTL formula is expressed over is modified.

When applied to concurrent programs, for efficiency reasons we assume interleavings only at statement boundaries and assume sequential consistency [114], but options to ESBMC allow us also to use a finer-grained analysis to detect data races arising from interleavings within statements.

3.1.1.3 Monitor threads

A *monitor* is some portion of code that inspects a program state and verifies that it satisfies a given property, failing an assertion if this is not the case. A monitor thread is a monitor that is interleaved with the execution of the program under test. This allows it to verify that the property holds at each particular interleaving of the program, detecting any transient violations between program interleavings.

Monitor threads have been employed in SPIN to verify LTL properties against the execution of a program [93]. A non-deterministic BA representing the negation of the LTL property, the so-called never claim, is implemented in a Promela process which will accept a program trace that violates the original LTL property. SPIN then generates execution traces of interleavings of the program being verified, and for each step in each trace runs the Promela BA. This is called a *synchronous interleaving*. In our work we employ a similar mechanism to verify LTL properties by interleaving the program under verification with a monitor thread as detailed in the next section.

A number of algorithms exist for converting an LTL formula to a BA accepting a program trace [82, 144, 90]. We use the `ltl2ba` [82] algorithm and tool, which produces smaller automata

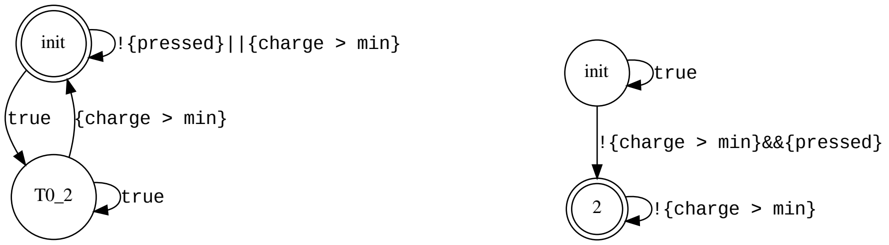

FIGURE 3.1: The left BA accepts the example formula from the introduction, `G({pressed} -> F {charge > min})`. The right BA is its negation, used for the never claim in our monitor

than some other algorithms [144]. Figure 3.1 illustrates the BA produced from the LTL formula in the introduction. Input symbols are propositions composed from the primitive C-expressions.

3.1.2 Checking LTL properties against a C program

As discussed in Section 3.1.1.3, an LTL property can be verified against a program by interpreting the corresponding BA over the program's states along its execution path. We apply this approach to a C code base by implementing the BA in C which is then executed as a monitor thread, interleaved with the execution of the program. This involves three technical aspects: the conversion of the BA to C, the interaction of the monitor thread with the program under test, and the control of the interleavings.

The monitor thread itself is not interleaved with the program in a special manner as in SPIN, but instead is treated as any other program thread. We use a counting mechanism to ensure that the BA thread operates on the program states in the right sequential order. This approach can be slower than a synchronous composition, but it requires no fundamental changes to the way that ESBMC operates, as it uses only existing features.

3.1.2.1 Implementing a Büchi automata in C

We follow the SPIN approach of inverting the LTL formula being verified so that the BA accepts execution traces which violate the original formula. We use a modified version of the `ltl2ba` tool to convert its usual Promela output to C.

Listing A.1 presents the C implementation of the negated BA shown in Figure 3.1). It consists of an infinite loop around a switch statement on a state variable, with the state variable valuations corresponding to a state in the BA (see line 6). For each BA state, it atomically (lines 18, 46) transitions to the next state of the BA. Non-deterministic behaviour is simulated by attempting all transitions from a state non-deterministically (lines 24, 27, 36), after which guards on each transition evaluate whether the transition can be taken (lines 25, 28, 37). These guards use

ESBMC's *assume* statements, which ensure that transitions not permitted by the current state of the program under test are not explored.

This ensures that ESBMC's symbolic execution of the original program drives the evolution of the BA through the possible states. The code for the BA is not actually, but only symbolically, executed, we do not have to model the non-determinism of the BA directly in the C code (e.g., by keeping a set of current states), and can instead represent the possible current states of the BA as a non-deterministic but properly constrained single integer variable. That is, the C code will transition only from one state to another, not from one set of states to another. We then rely on the model checker to explore all possible transitions. This makes good use of the capabilities of the SMT solver and substantially simplifies the implementation of the monitor.

To determine when the BA has accepted a program trace, we first await a time when the program has terminated—given that we operate in the context of bounded model checking this is guaranteed as any infinite loop is unrolled only to the length of the bound. Detection of thread deadlock has already been performed by ESBMC. When the program has terminated, the BA will have received each state of the program as input in the monitor thread. The BA loop is run a second time with the final program state as input, recording the number of times it passes through each state (lines 44-45). If a loop through an accepting state exists, it will be visited more than once, triggering an assertion showing that the BA accepted the trace. This technique places a constraint on the unwinding bound of the BA loop; it has to be sufficient for any such loop to be detected. Manually setting this bound to twice the number of states in the BA permits it to pass through every state twice on the largest possible loop.

This acceptance criteria operates on the principle that, should some program state need to be reached for the LTL formula to hold or to fail, then it needs to have happened by the time that the program bound has been reached. This can be an under-approximation as there can be circumstances where a violating program state could be reached if the program bound were higher.

We strictly control where interleavings may occur in the BA to ensure its soundness. The evaluation of the next state is executed atomically, ensuring that the BA always has a consistent view of program state. We also yield execution (line 17) before the BA inputs a program state so as to force new interleavings to be explored. Certain utility functions are provided to allow a program test harness to start the BA and check for acceptance at the end of execution (not shown).

3.1.2.2 Interacting with the existing code base

LTL formulae allow verification engineers to describe program behaviour using propositions about program states. To describe the state of a C program, we support the use of C expressions as propositions within LTL formulae. Any characters enclosed in curly brackets in the formula are interpreted as a C expression and as a single proposition within LTL. The expression itself may use any global variables within the program under analysis as well as constants and side-effect free operators. The expression must also evaluate to a value that can be interpreted as a truth value under conventional C semantics.

For example, the following liveness property verifies that a certain input condition results in a timer eventually increasing:

$$\mathsf{G}((\{\texttt{press == 4}\} \land \{\texttt{mstate == 1}\}) \implies \mathsf{F}\{\texttt{stime > refstime}\})$$

and the following safety property checks a buffer bound condition:

$$\mathsf{G}(\{\texttt{buffer\_size != 0}\} \implies \{\texttt{next < buffer\_size}\})$$

Within the BA (see Listing A.1 again) these C expressions are used to guard against invalid transitions being explored. We avoid using the expressions directly in the BA; instead ESBMC searches the program under verification for assignments to global variables used in the C expression, then inserts code to update an auxiliary Boolean variable corresponding to the truth of the expression (lines 2 and 4) immediately after the global is changed. In case multiple propositions update on the same variable, re-evaluations are executed atomically. All modifications are performed on ESBMC's internal representation of the program and do not alter the original code base.

This transformation does not, however, handle indirect assignments to variables, i.e., assignments through dereferencing pointers. None of our test cases (see below) perform such actions—in fact our application domain (embedded software) tends not to feature indirect operations at all, instead preferring to operate on a fixed set of configuration and data variables, due to memory and environment limitations. As a result we have not attempted to extend our approach to handle indirection. If required, it could be implemented through the use of a simple points-to static analysis to identify which global variables pointers may point at, and updating the relevant Boolean variables if a global is assigned through a pointer.

3.1.2.3 Synchronous Interleaving

A problem with operating the monitor thread containing the BA as a normal program thread is that it is not always guaranteed to receive a complete sequence of valuations—that is, it is entirely possible for the BA not to be scheduled to run after an event of interest, and thus not perform a state transition it should have. This is clearly broken, as the BA may see a trace with input characters artificially deleted. The full exploration of multithreaded state space guarantees that we will explore interleavings where this occurs, as well as interleavings where the monitor thread runs after every event of interest.

To address this, the BA discards interleavings where the propositions have changed more than once but the BA has not had opportunity to run and interpret them (lines 19–21 in Listing A.1).

We maintain a global variable (line 10) counting the number of times that the C expressions forming propositions in the LTL formula have been re-evaluated, keep a corresponding counter (line 9, 21) within the BA, and use an assume statement to restrict ourselves to traces where the global counter has changed at most once since the last time the BA ran. This ensures that the only interleavings considered are those where the BA runs every time the input symbol changes.

3.1.3 Experimental Evaluation

We have tested the work described here against a series of properties defining the behaviour of a pulse oximeter firmware, which is a piece of sequential software that is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method [67]. The firmware of the pulse oximeter device is composed of device drivers (i.e., display, keyboard, serial, sensor, and timer) that contain hardware-dependent code, a system log component that allows the developer to debug the code through data stored on RAM memory, and an API that enables the application layer to call the services provided by the platform. The final version of the pulse oximeter firmware consists of approximately 3500 lines of ANSI-C code and 80 functions.

The source code to the pulse oximiter is listed in Section A.2, with our test harnesses and any modifications made documented in Section A.3.

To improve the performance of this verification approach, we implemented an optimisation technique called *state hashing*, details of which are covered in Section 5.2.

Here we report the results of verifying the pulse oximeter code against five liveness properties taken from an SMV model of the software [69], of the general form `G(p -> F q)` i.e., whenever an enabling condition `p` has become true, then eventually the property `q` is true. We formulated a test harness for each portion of the firmware being tested to simulate the activity that the LTL property checks. We then invoked ESBMC with a variety of loop unwind and context switch bounds to determine the effectiveness of state hashing. We also ran these tests against versions of the firmware deliberately altered not to satisfy the LTL formula, to verify that failing execution traces are identified.

All tests were run on the Iridis 3 compute cluster[3] with a memory limit of 4Gb and time limit of 4 hours to execute. The results are summarized in Table 3.1. Here, the #L column contains the line count of the source file for the portion of firmware being tested, P/F records whether the test is expected to Pass or Fail, k the loop unwinding bound and C the context-bound specified for the test.

We report the results for the original version of ESBMC [4] and the version with state hashing, respectively. For each version, we report the verification time in seconds, the number #I and

[3] 1008 Intel Nehalem compute nodes, each with two 4-core processors, up to 45Gb of RAM, and InfiniBand communications. Each test used only one core of one node.

[4] v1.16, available from www.esbmc.org

					Original run			With state hashing		
Name	#L	P/F	k	C	Time	#I / #FI	Res	Time	#I / #FI	Res
start_btn	856	Pass	1	20	207	7764/0	+	67	2245/0	+
		Pass	1	40	199	7764/0	+	71	2245/0	+
		Pass	2	20	2740	55203/0	+	479	11409/0	+
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	236	6719/231	+	81	1919/91	+
		Fail	1	40	244	6719/231	+	94	1919/91	+
		Fail	2	20	1344	29840/0	−	299	6911/0	−
		Fail	2	40	N/A0	0/0	MO	N/A	0/0	MO
up_btn	856	Pass	1	20	78	3775/0	+	32	1385/0	+
		Pass	1	40	83	3775/0	+	37	1385/0	+
		Pass	2	20	2777	102566/0	+	898	41389/0	+
		Pass	2	40	14400	0/0	TO	6012	111335/0	+
		Fail	1	20	90	3775/0	−	35	1385/0	−
		Fail	1	40	82	3775/0	−	33	1385/0	−
		Fail	2	20	2743	102564/0	−	914	40938/0	−
		Fail	2	40	14400	0/0	TO	4832	69275/3422	+
keyb_start	50	Pass	1	20	9668	92795/0	+	4385	49017/0	+
		Pass	1	40	9767	92795/0	+	4489	49017/0	+
		Pass	2	20	14400	0/0	TO	14400	0/0	TO
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	9795	92795/321	+	4836	49017/321	+
		Fail	1	40	9924	92795/321	+	4914	49017/321	+
		Fail	2	20	14400	0/0	TO	14400	0/0	TO
		Fail	2	40	14400	0/0	TO	14400	0/0	TO
baud_conf	178	Pass	1	20	18	485/0	+	16	419/0	+
		Pass	1	40	17	485/0	+	16	419/0	+
		Pass	2	20	2440	39910/0	+	971	17500/0	+
		Pass	2	40	2635	39910/0	+	1078	17500/0	+
		Fail	1	20	18	485/56	+	17	419/56	+
		Fail	1	40	18	485/56	+	16	419/56	+
		Fail	2	20	2583	39910/2002	+	1010	17500/880	+
		Fail	2	40	2851	39910/2002	+	1139	17500/880	+
serial_rx	584	Pass	1	20	334	5454/0	+	194	3108/0	+
		Pass	1	40	324	5454/0	+	212	3108/0	+
		Pass	2	20	10959	62332/0	+	4494	29257/0	+
		Pass	2	40	14400	0/0	TO	70	627/0	+
		Fail	1	20	215	3286/273	+	137	2030/257	+
		Fail	1	40	211	3286/273	+	135	2030/257	+
		Fail	2	20	3768	20917/0	−	1846	11388/0	−
		Fail	2	40	14400	0/0	TO	14400	0/0	TO

TABLE 3.1: Results from testing LTL properties against pulse oximeter firmware. Time is given in seconds

#FI of generated and failing interleavings, respectively, and the result. Here, "+" indicates that ESBMC's result is as expected (i.e. all its interleavings were verified successfully if the test is expected to pass, and at least one interleaving is found to violate the LTL property if the test is expected to fail), while "−" indicates a false negative (i.e., ESBMC fails to find an existing violation of the LTL property). TO indicates the check ran out of time and MO indicates it ran out of memory.

We first observe that ESBMC is generally able to verify all positive test cases, although it tends to time out with increasing bounds. The situation is less clear for the tests designed to fail. Here, smaller unrolling and context switch bounds allow us to correctly identify failing interleavings, but are sometimes not sufficient to expose the error (e.g., up_btn), and small increases in the unrolling bound generally require larger increases in the context bounds to expose the error, leading to time-outs or memory-outs in most cases. The state hashing optimisation, however, improves the situation, and allows us to find even deeply nested errors.

3.1.4 Analysis

Using context-bounded model checking to check LTL properties of ANSI-C software appears to be an effective approach, finding the correct answers for each of the LTL formulae tested in the experiments, for safety and liveness formula. In particular, the ability to test formulae against the codebase without modification (save the necessary external test harness) makes verification simple for the engineer. There are however, two significant limitations to the approach.

Firstly, the indiscriminate composition of the monitor thread with the program under test leads to a very large number of interleavings that need to be explored. This necessitated the addition of the aforementioned state hashing optimisation, which moderately reduced the state explosion effect. It is particularly awkward for single threaded programs, which do not naturally experience state explosion from concurrency, but still pay the performance penalty from being composed with the monitor thread. Most problematic is that the majority of the interleavings produced were rejected by the assumptions added in Section 3.1.2.3, where the monitor thread is not scheduled sufficiently frequently. The reason behind selecting this method was simply to avoid in-depth modifications of the model checker. It is clear, however, that further improvements must be made for the sake of performance.

Secondly, the stutter extension discussed in Section 3.1.1 can make the checking of certain formula meaningless. Consider a co-safety property (Section 2.3) and a piece of code being checked that the property holds over, but only over particularly long traces, that would require a large unwinding bound to reach. Such a co-safety would be reported as a verification failure by our approach for any unwinding bound lower than the one required, as the formula would not hold on the infinite stutter extension of a co-safety property that has not been fulfilled, despite the fact that the formula eventually holds on the program. This result is at odds with the aims of bounded model checking, to be sound but not complete, as the result reports an unsound property violation. Safety properties that hold over the finite prefix will be reported as holding over the whole execution of the program, despite future states possibly violating the property.

P_1:

```
int s=0;
while(true){
  s=1-s;
};
```

P_2:

```
int s=0;
while(true){
  s=1;
  s=0;
};
```

P_3:

```
int s=0;
s=1;
while(true){
  s=0;
  s=1;
};
```

FIGURE 3.2: Programs with identical infinite traces but different behaviour on finite unwindings for $\gamma \equiv \mathsf{G}(\{\texttt{s=0}\} \implies \mathsf{F}\{\texttt{s=1}\})$.

Infinite stutter extension coupled with partial loop evaluation can also cause the validity of a formula to be decided by the structure of the program under test rather than the program's validity. Consider the code samples in Figure 3.2 and the liveness property $\gamma \equiv \mathsf{G}(\{s == 0\} \implies \mathsf{F}\{s == 1\}$,. The single infinite trace of states produced by each program is identical, alternating between the states $\{\texttt{s==0}\}$ and $\{\texttt{s==1}\}$; γ holds over this trace. Were we to stutter extend any of the finite prefixes of the trace, the trace would latch in one of those states forever, which is to be expected. This would, however, make the truth of γ depend on what finite prefix we select.

Intuitively, P_3 will always be reported as a verification success, as any number of loop unwinding bounds will end in a state where `s==1`, the stutter extension of which causes the corresponding BA (refer back to Figure 3.1 on page 55) to either reject or loop infinitely in a non-accepting state. The opposite will occur for P_2 though, as it terminates in a state where `s==0`. P_1 will behave erratically as the state it terminates in depends entirely on the number of loop unwindings the program was permitted. This range of results that can be caused by the structure of the program, in combination with stutter extension, significantly reduces confidence in this approach.

3.2 Improved LTL model checking of bounded traces

Given the drawbacks of our model checking approach highlighted in the previous section, we studied several ways of improving the performance of our approach and the accuracy of the violations that it reports. Our work here was originally presented in [128].

To address the performance issues associated with composing the program under test with a monitor thread, we instead schedule the monitor thread using knowledge about the changes to property variables, manually generating the synchronous interleaving referred to in Section 3.1.2.3. This ensures that state explosion caused by concurrent threads is not amplified by the monitor thread. The cost of evaluating the monitor's BA is, however, still present in the SMT formula solved at the end of each finite trace produced.

We also extend the truth values of the LTL expressions to a four-valued lattice describing the least truth values over various possible future behaviours of a C program with possibly infinite state space. We consider the explored traces to be finite prefixes of infinite traces and our four-valued logic describes the accepting behaviour of the BA for different infinite extensions of the explored finite traces. In practice, the never claim BA obtained from commonly used specifications tends to have a small number of states. The small size allows us to analyse which states are accepting under the different infinite extensions of the finite traces. We then check the combined system several times, with different assertions corresponding to the different acceptance criteria, to derive the correct truth value for the LTL formula. The program's overall "correctness" value in the lattice is the weakest truth value for which the model checker can find a witness that violates the corresponding assertion. This gives us a method to analyse both safety and liveness within the framework of bounded software model checking.

3.2.1 Multi-valued LTL

The principles of LTL formulae with extended values have already been explored elsewhere. We follow the exposition by Bauer et al. [27] and use finite deMorgan lattices as truth domains. A *deMorgan lattice* is a distributive lattice $(\mathcal{L}, \sqcup, \sqcap, \top, \bot)$ where every element $x \in \mathcal{L}$ has a *dual* element $\overline{x} \in \mathcal{L}$ such that $\overline{\overline{x}} = x$ and $x \sqsubseteq y$ implies $\overline{y} \sqsubseteq \overline{x}$; here, $\sqsubseteq$ is the partial order induced by the lattice structure. Note that not every deMorgan lattice is a Boolean lattice, because duals are not proper complements (i.e., $x \sqcap \overline{x} = \bot$ is not necessarily true), but the converse holds, and in particular the Boolean lattice over the standard two-valued truth domain $\mathbb{B}_2 = \{\bot, \top\}$ is a deMorgan lattice with $\bot \sqsubseteq \top$.

We can then define the standard semantics of LTL formulae via the interpretation function $[\_ \models \_]_\omega : \Sigma^\omega \times LTL \to \mathbb{B}_2$, as shown in Figure 3.3 [27]. We call the trace $w \in \Sigma^\omega$ a *model* of the LTL formula φ iff $[w \models \varphi]_\omega = \top$ and also say that w satisfies φ, or that φ holds for w. For each LTL formula the set of all its models is an ω-regular language that is accepted by a corresponding Büchi automaton [153, 154].

We interpret a possibly multi-threaded C program P as a Kripke structure in the manner given in Section 3.1.1. P can be non-deterministic, so the transition relation can branch even for single-threaded programs. As C's semantics gives a defined (zero) value to all global variables not initialised explicitly at their declaration, all valuations are completely defined in every state. We identify a C program P with the set of all traces $\mathcal{T}(P)$ that correspond to this Kripke structure, and say that an LTL formula φ holds for P if φ holds for all $w \in \mathcal{T}(P)$.

3.2.2 LTL over Finite Traces

Our aim here is to extend LTL to be well defined on finite traces, so that more information can be gleaned about the way in which a formula holds on a system. The fundamental problem is that the standard interpretation of X is a strong (or existential) next operator [107], which requires the existence of a next state to hold. This is counter-intuitive for finite traces, since $\mathsf{X}\,\mathsf{true}$ is

Propositional constants.

$$[w \models \mathsf{true}]_\omega = \top \qquad [w \models \mathsf{false}]_\omega = \bot \qquad [w \models p]_\omega = \begin{cases} \top & \text{iff } p \in w_0 \\ \bot & \text{iff } p \notin w_0 \end{cases}$$

Propositional operators.

$$[w \models \varphi \vee \psi]_\omega = [w \models \varphi]_\omega \sqcup [w \models \psi]_\omega \qquad [w \models \neg\varphi]_\omega = \overline{[w \models \varphi]_\omega}$$

Temporal operators.

$$\begin{aligned}
[w \models \mathsf{X}\varphi]_\omega &= [w^1 \models \varphi]_\omega \\
[w \models \mathsf{F}\varphi]_\omega &= \begin{cases} \top & \text{iff } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ \bot & \text{otherwise} \end{cases} \\
[w \models \mathsf{G}\varphi]_\omega &= \begin{cases} \top & \text{iff } [w^i \models \varphi]_\omega = \top \text{ for all } i \geq 0 \\ \bot & \text{otherwise} \end{cases} \\
[w \models \varphi \mathsf{U} \psi]_\omega &= \begin{cases} \top & \text{iff } [w^i \models \psi]_\omega = \top \text{ for some } i \geq 0 \\ & \quad \text{and } [w^j \models \varphi]_\omega = \top \text{ for all } 0 \leq j < i \\ \bot & \text{otherwise} \end{cases} \\
[w \models \varphi \mathsf{R} \psi]_\omega &= \begin{cases} \top & \text{iff } [w^i \models \psi]_\omega = \top \text{ for all } i \geq 0 \\ & \quad \text{or } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ & \qquad \text{and } [w^j \models \psi]_\omega = \top \text{ for all } 0 \leq j \leq i \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

FIGURE 3.3: Standard LTL semantics over infinite traces.

now no longer a tautology, as $\models_F$ (i.e., the standard interpretation applied to finite traces) gives us, for all formulae φ, $[u \models \mathsf{X}\varphi]_F = \bot$ if $u^1 = \epsilon$ [27].

Several approaches tweak the syntax or semantics of LTL to remedy this situation. Since G and F can be defined relatively straightforwardly on finite traces, Giannakopolou and Havelund [83] suggested removing X and working with an X-free subset of LTL. The syntax can instead be extended by adding an additional weak (or universal) next operator $\overline{\mathsf{X}}$ [118], which complements the strong next and holds if there is no next state: $[u \models \overline{\mathsf{X}}\varphi]_F = \top$ if $u^1 = \epsilon$. Hence, $\overline{\mathsf{X}}\,\mathsf{true}$ *is* a tautology. This also gives unwinding laws for F and G, namely $\mathsf{F}\varphi \equiv \varphi \vee \overline{\mathsf{X}}\mathsf{F}\varphi$ and $\mathsf{G}\varphi \equiv \varphi \wedge \overline{\mathsf{X}}\mathsf{G}\varphi$. Alternatively, the distinction between strong and weak next can be encoded into the semantics rather than the syntax, via two different semantic functions which coincide on the temporal and most Boolean operators, but differ on negation (which flips between both functions) and the atomic propositions, where they reflect the behaviours of strong and weak next, respectively [78]. Bauer et al. conclude that these approaches are neither *impartial* nor *anticipatory*—they can prematurely conclude that a formula holds, and may not conclude a formula holds even when sufficient information is available [27].

The crux of the matter is that in a two-valued logic we cannot distinguish between a formula that (truly) holds because we have seen a *good prefix* [112] and so **all** possible continuations of the observed finite trace will be models as well, those that cannot hold because we have seen a *bad prefix* (i.e., a finite trace that **cannot** be the prefix of a model), and those that (presumably) hold because we have not yet seen a bad prefix.

In order to realise this distinction, we use a larger truth domain. Bauer et al. [26, 27, 28] have proposed and analysed two different domains, $\mathbb{B}_3 = \{\bot, ?, \top\}$, with $\bot \sqsubseteq ? \sqsubseteq \top$, $\overline{\bot} = \top$, and $\overline{?} = ?$, and $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$, with $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$, $\overline{\bot} = \top$, and $\overline{\bot^p} = \top^p$. Under $\models_3$, finite traces are mapped to $\top$ (resp. $\bot$) iff they are good (resp. bad) prefixes; all other finite traces are considered "ugly" and are mapped to the inconclusive truth value ? [26, 28]. In $\mathbb{B}_4$, ? is refined into the two truth values $\bot^p$ ("presumably false") and $\top^p$ ("presumably true"). The interpretation function $\models_4$ then uses the finite trace semantics with weak next to distinguish between the two cases (i.e., $[u \models \varphi]_4 = \bot^p$ iff u is an ugly prefix and $[u \models \varphi]_F = \bot$, and similarly for $\top^p$) [27].

Our analysis here is based on $\mathbb{B}_4$ as well, but we use a different interpretation function from Bauer et al. [27]. We use the infinite extension semantics discussed in the previous section to resolve ugly prefixes into presumably good or presumably bad, whereas Bauer et al. classify ugly prefixes using the weak-next operator. The advantage of our approach is that we can define the finite trace semantics in terms of the standard semantics only.

Definition 3.1. The *bounded trace semantics* of LTL formulae is given by

$$[u \models \varphi]_B = \begin{cases} \top & \text{iff} \quad \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \top^p & \text{iff} \quad [uu_{n-1}^\omega \models \varphi]_\omega = \top \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \\ \bot^p & \text{iff} \quad [uu_{n-1}^\omega \models \varphi]_\omega = \bot \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \bot & \text{iff} \quad \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \end{cases}$$

for a finite trace $u \in \Sigma^*$ of length $n > 0$ and an LTL formula φ.

In our case, all program traces are guaranteed to be non-empty, because all global variables have defined initial values, which then form the initial state. We extend the interpretation to sets of traces by taking the meet over all elements, i.e., $[U \models \varphi]_B = \bigsqcap_{u \in U} [u \models \varphi]_B$. We say that φ *holds* (resp. *presumably holds*) *for a C program P* if $[\mathcal{T}(P) \models \varphi]_B = \top$ (resp. $\top^p$). We finally say φ *holds* (resp. *presumably holds*) if $[\Sigma^\omega \models \varphi]_B = \top$ (resp. $\top^p$) and define the notion of *fails* resp. *presumably failing* correspondingly.

Considering again the example programs in Figure 3.2, checking the value of γ over P_2 will now result in $\bot^p$, indicating that a *bad* trace has not been seen but that the program terminates in a state that fails when stutter extended, and P_3 results in $\top^p$, indicating γ holds on the stutter extension of the final state but a good prefix has not been seen. These results allow us to identify whether the program terminates in a state that places an obligation on future states for the formula to hold. Analysis of the possible values of various LTL formulae is discussed in Section 3.2.5.

3.2.3 LTL Model Checking vs. LTL Runtime Verification

Finite LTL semantics similar to the bounded trace semantics we are using here have been developed largely for run-time monitoring and verification purposes [116], and due to the focus on finite traces, our approach has some similarities with run-time verification, but one key difference remains. Runtime verification only considers actual observed behaviours, one at a time, while we analyse all possible behaviours in the same run. This difference becomes prominent with non-determinism, even for single-threaded programs. Consider for example the program Q

```
int p=0, q=0; p=1; if(*){p=0}; if(*){q=1};
```

where "$\star$" denotes a non-deterministic choice and p and q are zero-initialised global variables. Q can produce four distinct stutter-free finite traces, depending on the particular non-deterministic choices. I present them here as sequences of states, separated by ",":

(i) $[\{\texttt{p==0}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==1}\} \wedge \{\texttt{q==0}\}]$,

(ii) $[\{\texttt{p==0}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==1}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==1}\} \wedge \{\texttt{q==1}\}]$,

(iii) $[\{\texttt{p==0}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==1}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==0}\} \wedge \{\texttt{q==0}\}]$, and

(iv) $[\{\texttt{p==0}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==1}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==0}\} \wedge \{\texttt{q==0}\}, \{\texttt{p==0}\} \wedge \{\texttt{q==1}\}]$.

Now consider the LTL formula $\psi \equiv \mathsf{G}(\{\texttt{p==1}\} \implies \{\texttt{p==1}\} \,\mathsf{U}\, \{\texttt{q==1}\})$. Clearly, ψ does not hold for the traces (iii) and (iv), and over these, $\models_3$, $\models_4$, and $\models_B$ all map ψ to $\perp$. However, in run-time verification, there is no guarantee that we ever observe these traces, so the assurance we gain from its results is limited. Our approach, however, will work out that $[\mathcal{T}(Q) \models \psi]_B = \perp$ and hence Q can fail ψ. Moreover, if we consider Q' to be the variant of Q where q is initialised with one, we find $[\mathcal{T}(Q') \models \psi]_B = \top$ as well. Finally, if we change Q to Q''

```
int p=0, q=0; p=1; if(*){q=1};
```

then (iii) and (iv) become impossible, and our approach will calculate $[\mathcal{T}(Q'') \models \psi]_B = \perp^p$, meaning that no finite trace produced by Q'' is a definitive counter-example but, on stuttering, ψ does not hold for all traces.

3.2.4 Characterising Program Behaviours Using $\mathbb{B}_4$

Definition 3.1 characterises the truth value in $\mathbb{B}_4$ of an LTL formula φ with respect to a single finite trace u. In this section we now show how we can use the Büchi automaton for the never claim to effectively calculate the truth value of the formula with respect to the finite traces of a program P. In Section 3.2.4.1, we identify characteristics of the BA that we can use to classify input traces. In Section 3.2.4.2 we characterise the relationship between truth values in $\mathbb{B}_4$ and validity of never claims over $\mathbb{B}_2$, while we describe the high-level structure of our algorithm in Section 3.2.4.3.

3.2.4.1 Büchi Automata characteristics

Büchi automata (BA) were introduced in Section. 2.3.2. Here we chose to deal with BA's that are in *reduced form* [14], meaning they have no rejecting traps, i.e., there are no transitions to states where no extension of the trace accepts. This reduction is conveniently already performed by the `ltl2ba` [82] algorithm tool that we use to produce BA's. An example of such a BA, in it's positive and negative forms, is in Figure 3.1, corresponding to the formula $\mathsf{G}(\{\texttt{pressed}\} \implies \mathsf{F}\{\mathit{charge} > \mathit{min}\})$. The reduction can been seen in that, for the never-claim BA, the accepting state (labelled "2") has no available transition if `charge > min` is true. Without the reduction, there would be a transition from "2" to a non-accepting state with a single self looping `true` transition, representing a state that traps the BA to never accept.

We can apply a simple reachability static analysis to determine, for a given BA state and program state, whether stutter extension of the program state would lead to an accepting loop through the BA. Extending this, we can then determine whether there are *any* program states that can cause the BA to reject once it has reached a particular state, and likewise whether there are any program states that can lead to an accepting loop. This allows us to identify (for each BA state) whether we have observed a good prefix where all future program states lead to the BA rejecting, or a bad prefix where acceptance is inevitable.

Rather than using a complicated algorithm, we instead enumerate all BA states and all inputs characters, and explore all further reachable states for each combination of the two. States where no input leads to an accepting loop are identified as being part of a good prefix, while states where all inputs lead to an **unconditional** accepting loop are identified as being part of a bad prefix. States that meet neither of these criteria have their stutter acceptance analysed: we explore the reachability of all states assuming one input is infinitely repeated, and evaluate whether an accepting loop is reachable. The results of each of these tests are used to produce the monitor code discussed in Section 3.2.6.

Our reachability analysis grows exponentially with both propositions and BA states, which limits us to working only on small BAs.

3.2.4.2 Truth Values in $\mathbb{B}_4$ and Standard Validity of Never Claims

As noted above, Definition 3.1 characterises the truth value in $\mathbb{B}_4$ of an LTL formula φ with respect to a single finite trace u. However, for model checking φ over a program P this is not yet suitable. First, we need to express the truth value in $\mathbb{B}_4$ in terms of the validity of the never claim under the two-valued standard semantics. This allows us to use the BA for the never claim directly, and avoids the need to define an explicit acceptance criterion for the four-valued logics. The following lemma addresses this problem. Note that we do not need a complete characterisation of all truth values in $\mathbb{B}_4$.

Lemma 3.2.

$$
\begin{array}{llll}
(i) & [u \models \varphi]_B = \top & \text{iff} & \nexists w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top \\
(ii) & [u \models \varphi]_B \sqsupseteq \top^p & \text{iff} & [uu_{n-1}^\omega \models \neg\varphi]_\omega = \bot \\
(iii) & [u \models \varphi]_B = \bot & \text{iff} & \forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top
\end{array}
$$

Proof. (i) Since the standard semantics $\models_\omega$ (cf. Figure 3.3) is defined over $\mathbb{B}_2$, $\nexists w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \bot$, and thus to $\forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top$, which gives us the claim.
(ii) Similarly, $[uu_{n-1}^\omega \models \neg\varphi]_\omega = \bot$ is equivalent to $[uu_{n-1}^\omega \models \varphi]_\omega = \top$, which holds if and only if $[u \models \varphi]_B = \top$ or $[u \models \varphi]_B = \top^p$.
(iii) This follows directly from the definitions of $\models_\omega$ and $\models_B$. $\square$

Second, the program P may be non-deterministic or multithreaded, and produce more than one trace. We thus need to consider the minimum truth value attained over all of its possible traces $\mathcal{T}(P)$. The following lemma addresses this problem.

Lemma 3.3.

$$
\begin{array}{llll}
(i) & [U \models \varphi]_B = \top & \text{iff} & \nexists u \in U, w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top \\
(ii) & [U \models \varphi]_B \sqsupseteq \top^p & \text{iff} & \nexists u \in U \cdot [uu_{n-1}^\omega \models \neg\varphi]_\omega = \top \\
(iii) & [U \models \varphi]_B = \bot & \text{iff} & \exists u \in U \cdot \forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top
\end{array}
$$

Proof. Recall that $\bigsqcap_{u \in U} [u \models \varphi]_B = [U \models \varphi]_B$. Then:
(i) By Lemma 3.2 $\nexists u \in U, w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\forall u \in U \cdot [u \models \varphi]_B = \top$; hence, $[U \models \varphi]_B = \top$.
(ii) By definition of $\models_\omega$, $\nexists u \in U \cdot [uu_{n-1}^\omega \models \neg\varphi]_\omega = \top$ is equivalent to $\forall u \in U \cdot [uu_{n-1}^\omega \models \varphi]_\omega = \top$, which by definition of $\models_B$ means that $\forall u \in U \cdot [u \models \varphi]_B \sqsupseteq \top^p$, and thus $[U \models \varphi]_B \sqsupseteq \top^p$.
(iii) By the definitions of $\models_\omega$ and $\models_B$ we have that $\exists u \in U \cdot \forall w \in \Sigma^\omega \cdot [uw \models \neg\varphi]_\omega = \top$ is equivalent to $\exists u \in U \cdot [u \models \varphi]_B = \bot$ and thus $[U \models \varphi]_B = \bot$ as well. $\square$

3.2.4.3 Algorithm Structure

Lemma 3.3 rephrases the definition of validity in $\mathbb{B}_4$ into a form that is suitable for model checking a program against a standard non-deterministic never claim BA. In particular, in all but the inner clause of the test for $\bot$ the quantifiers are existential and are thus compatible with the existential (i.e., optimistic) search for accepting traces.

In the following we use BA$_{\neg\varphi}$ to denote the never claim BA for the LTL formula φ. We also assume that BA$_{\neg\varphi}$ is in reduced form [14]. These assumptions make the application of the tests below straightforward.

$[\mathcal{T}(P) \models \varphi]_B = \top$: As BA$_{\neg\varphi}$ is in reduced form, it cannot accept the program trace if it has no transition from its set of current states to a next state, and the trace can be pruned. If and only if all traces are pruned, the program evaluates to $\top$. Note that this cannot happen [14] if φ is a (non-trivial) classical safety property [13].

$[\mathcal{T}(P) \models \varphi]_B = \bot$: If BA$_{\neg\varphi}$ reaches an accepting trap for any trace, φ evaluates to $\bot$ over the program, with the trace returned as a witness. Note that this cannot happen [14] if φ is a classical liveness property [13].

$[\mathcal{T}(P) \models \varphi]_B = \top^p$: If the property does not evaluate to $\top$ or $\bot$, we check its stutter acceptance. The simple static analysis of the BA from Section. 3.2.4.1, given the transitions enabled in the final program state, allows us to check for possible stutter acceptance at the end of each symbolically generated set of traces. If no accepting cycle is found, the property evaluates to $\top^p$, with one of the traces returned as a witness.

$[\mathcal{T}(P) \models \varphi]_B = \bot^p$: If BA$_{\neg\varphi}$ stutter accepts for at least one trace, the property evaluates to $\bot^p$ and the trace is returned as witness.

Note that the different cases are not independent of each other, due to the inequality in Lemma 3.3 (ii). As we are looking for a witness to the worst bounded behaviour that the program can exhibit when we model check, we must check multiple cases, although in the implementation of the algorithm (cf. Section 3.2.6) we check in a specific order to avoid redundant checks.

3.2.4.4 Example

As an example, consider the BA on the right of Figure 3.1, i.e., the never claim BA for the formula `G({pressed}` $\implies$ `F{charge > min})`. This BA is generated by `ltl2ba` and is already optimised, and in reduced form. Hence, it can accept on *some* infinite suffix from any state, and the set of optimistically accepting states is {init, 2}. There is no explicit trap state and thus, as this is an optimised BA, the set of states which will accept for all infinite suffixes is empty. The interesting behaviour of this request-response liveness condition is, as explained further in Section 3.2.5.3, restricted to its behaviour on infinite stutter. There are four possible infinite stutter suffixes and their accepting sets are shown in Table 3.2. Hence, if `{charge > min}` and `{pressed}` are both false in the final program state, the BA stutter accepts only if it is in state 2, and thus the trace is presumably failing only then.

Final symbol	Stutter-accepting states
$\neg$\{`charge > min`\} $\wedge$ $\neg$\{`pressed`\}	\{2\}
$\neg$\{`charge > min`\} $\wedge$ \{`pressed`\}	\{init, 2\}
\{`charge > min`\} $\wedge$ $\neg$\{`pressed`\}	$\emptyset$
\{`charge > min`\} $\wedge$ \{`pressed`\}	$\emptyset$

TABLE 3.2: Final symbol valuations and their corresponding stutter-accepting states.

3.2.5 Checking Safety, Co-Safety, and Liveness Properties

The previous section has described our process for evaluating whether LTL properties hold on a software system, with the result a member of $\mathbb{B}_4$. While the members of $\mathbb{B}_4$ have an order of "how correct" they are, a given property may only be able to evaluate to a subset of $\mathbb{B}_4$, regardless of the traces observed in the system under test. This makes it difficult to have confidence that a formula holds over a system as it might be unclear what responses to expect from ESBMC.

To address this, here we explore some of the different ways that LTL formulae can be classified, and what results ESBMC can produce for that particular classification of formula. We use the reachability static analysis of the never-claim BA and its knowledge of whether an accepting state is reachable, to determine for any LTL expression, which of the four elements of $\mathbb{B}_4$ can be returned, allowing us to estimate infinite or long-time program behaviours from the data returned by ESBMC. We are, therefore, able to distinguish safety, co-safety, "true" liveness and "toggle" liveness properties and thus to guide the expectations of the ESBMC user.

3.2.5.1 Safety Properties

In an imperative language such as C, it is common to test the validity of safety or invariant properties at various points in the program execution via `assert`-statements. These may be checked during program execution using the standard C library and, in conjunction with a suitable test suite, allow checking a variety of runs of the code as noted in Section 3.2.3. They are also recognised and checked during symbolic execution by ESBMC which gives an exhaustive examination of their validity for all (bounded) execution traces. Thus the code fragment on the left of Figure 3.4 will be verified successfully as the loop invariant `i+j==count` holds whenever the `assert`-statement is executed. It is, however, often more convenient to verify that a safety property holds everywhere *except* within a specific region in which updates are taking place, rather than just at particular locations. This is particularly attractive in languages such as C with limited support for data encapsulation: data that would be considered a private instance field in an object-oriented language is modifiable in C by a library's clients.

The classical safety property $\mathsf{G}\varphi$ states that φ must hold throughout program execution. However, this is of little practical use as it stands, because φ will typically be violated by any changes to its individual variables. Instead we model the permitted region in which the individual variables can be updated using a global flag `looking` which we set to zero during an update, and

```c
const int count=6;
int i=count;
int j=0;
...
while(i) {
  i--;
  j++;
  assert(i+j==count);
}
```

```c
const int count=6;
int i=count;
int j=0;
int looking=1;
  /* visibility to monitor */
...
while(i) {
  looking=0;
  i--;
  j++;
  looking=1;
}
```

FIGURE 3.4: C program with a safety assertion (left) and a monitor variable for a guarded safety property (right).

use a guarded safety property $\mathsf{G}(\{\texttt{looking}\} \implies \{\texttt{i+j==count}\})$. The listing on the right of Figure 3.4 shows the modified fragment together with the auxiliary code. In this case, the symbolic execution runs to completion and ESBMC reports $\top^p$.

Since it is in principle always possible for a safety property to be violated at some future time, no finite execution will cause the never claim BA to reject a word outright. Thus we can expect $\bot$ if the property is ever violated, and $\top^p$ if it holds for the trace prefix we have examined. If the program terminates, however, then we know no future violation can occur, and the the property holds. Our approach will stutter-extend the final state of the program and report $\top^p$, requiring us to verify that the program terminates by some other means, for example through the use of unwinding assertions (Section 2.4.2).

We can instead modify our LTL specification to capture explicitly the termination of the program; this is a natural use for the U operator. We simply add a second auxiliary variable `done` to capture program termination; this is initialised to zero, and set to one immediately before the program finishes. We then use the LTL specification $(\{\texttt{looking}\} \implies \{\texttt{i+j==count}\})\mathsf{U}\{\texttt{done}\}$. In this case, ESBMC reports a successful verification (i.e., $\top$) because the never claim BA fails; the invariant holds until `done` becomes true.

Note that, while accurately expressing a safety property over a terminating program, the second LTL expression does not meet the classical definition of a safety property [13] as finite prefixes can guarantee rejection of the never claim.

3.2.5.2 Co-Safety Properties

Co-safety properties [28] often reflect convergence or termination conditions. They are the converse of safety properties; they can be demonstrated to be true by some finite trace. Technically, they are a subset of liveness properties [13] as, whatever the initial trace, there is some future extension that can satisfy them. A co-safety property can never evaluate to $\bot$ in $\mathbb{B}_4$.

If we work again from the example shown in Figure 3.4, then the LTL formula $\mathsf{F}\{\texttt{j==6}\}$ expresses the termination (co-safety) condition that `j` will eventually reach its final value. When the program runs to completion, the condition is satisfied and ESBMC reports successful verification (i.e., $\top$). If we artificially restrict the number of loop interactions by setting the ESBMC flag[5] `--unwindset 1:4` to restrict the program loop to four iterations, ESBMC reports "presumably bad" (i.e., $\bot^p$). This is typical of a co-safety property; a gradually extended partial trace will continuously report "presumably bad" (as the necessary event has not happened) until it reports successful verification.

3.2.5.3 True Liveness Properties

Safety and co-safety properties have natural definitions over both finite and infinite traces, i.e., for terminating and for non-terminating programs. In contrast, *true* liveness properties[6] are generally regarded as well-defined only over infinite words. It is thus a challenge to use a bounded model checker to explore the true liveness properties of a program.

One of the simplest true liveness properties is a request-response formula of the form $\mathsf{G}(\varphi \implies \mathsf{F}\psi)$. The program is always required to respond to the request φ by producing a response ψ. We may examine this behaviour with the simple program

```
unsigned int i=0; int main() { while(1) i++; };
```

and the property $\mathsf{G}(\{\texttt{i\%2==0}\} \implies \mathsf{F}\{\texttt{i\%3==0}\})$. This property has the typical feature of a true liveness property: no finite trace can determine acceptance or rejection. A simple static analysis which searches for rejecting and accepting traps in the never claim BA already shows that this formula will (regardless of the program) never result in a definitive outcome (i.e., $\bot$ or $\top$).

In general, the regular appearance of $\top^p$ as we extend the length of the investigated prefix trace is characteristic of good programs under a request-response liveness property, while bad programs will eventually stop yielding in $\top^p$. This comes from the fact that a good program will regularly satisfy the response condition and evaluate to $\top^p$ at that point, while a bad program will never satisfy the response condition and will always respond $\bot^p$ once the request condition becomes true. For this example program, as we progressively increase the unwind bound from 1 to 12, the program's behaviour oscillates between $\top^p$ and $\bot^p$, and ESBMC reports:

$$\top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p, \top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p$$

In this particular case, we only have to bound a single loop in the program, making the trace seen a finite prefix of the programs infinite trace, with the size of the trace increasing as we increase

[5] In this context, the "1" identifies a specific loop, in this case the main program loop, and the "4" gives a specific bound to apply to that loop

[6] The classical definition of liveness properties [13] includes co-safety properties as well. Here we use the term *true liveness property* to exclude co-safety properties.

the unwind bound. More general programs can be more difficult to examine; if, for example, we have to bound several loops, the finite traces we observe may not even be valid prefixes of the real program behaviour. Nevertheless, in well-designed programs, we might hope that loop iterations would independently meet request-response liveness conditions and, as we increase the unwind bounds on the various loops we would expect to see regular appearances of $\top^p$.

A variant of the request-response liveness formula is often used as a fairness formula. The formula $\mathsf{GF}\{\texttt{p}\}$ expresses that the C expression `p` is true infinitely often at all times in the future. Such conditions can, for example, be conjoined into expressions of the form $(\bigwedge_i \mathsf{GF}\,\rho_i) \implies \mathsf{G}(\varphi \implies \mathsf{F}\psi)$ which are easily handled by our tools. Note that such expressions were the original motivation for the development of the compact BAs produced by `ltl2ba` [82].

Some liveness properties are resistant to an analysis with finite traces. "Toggle" properties such as $\mathsf{G}((\varphi \implies \mathsf{F}\neg\varphi) \wedge (\neg\varphi \implies \mathsf{F}\varphi))$ can be seen from our static analysis to have no stutter-accepting prefixes. The static analysis of the never claim BA for this formula shows that it responds with $\perp^p$ to all (non-empty) finite traces. Unfortunately, our tools are of little further use in this case, other than to confirm the impossibility of the task set in front of them. Thus, checking the formula

$$\mathsf{G}((\{\texttt{i\%2}\} \implies \mathsf{F}\neg\{\texttt{i\%2}\}) \wedge (\neg\{\texttt{i\%2}\} \implies \mathsf{F}\{\texttt{i\%2}\}))$$

over the above program, as we progressively unwind we see

$$\perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \perp^p, \ldots$$

3.2.5.4 Restricted Alphabets

Some symbols of the alphabet $\Sigma = 2^{Prop}$ cannot arise during program execution; this can happen if the various propositions are not independent. As an obvious example, consider a formula which includes both $\{\texttt{p}\}$ and $\{\texttt{!p}\}$ as primitive C expressions rather than negating in the LTL using $\{\texttt{p}\}$ and $\neg\{\texttt{p}\}$. This causes no problems with the evolution of the BA during program execution, nor with the computation of stutter-acceptance or rejection for $\perp^p$ or $\top^p$. Our system will, however, explore too large a symbol space when analysing for acceptance over all, or over no, future continuations. We might, in such situations, report $\perp_p$ where a more sensitive analysis would report $\perp$. ESBMC can itself be used, if necessary, to confirm the independence of the C expressions.

3.2.6 Implementation

Our implementation of the monitor thread broadly follows our previous approach, as described in Section 3.1.2.1, where the never claim is produced as a BA, and its structure replicated in a

C finite state machine, with nondeterministically taken transitions guarded with `assume` statements. Listing A.2 shows the C implementation of a monitor, with the never claim BA in Figure 3.1 (see page 55) contained in the function `ltl2ba_fsm` (lines 9 to 39). We require that the test harness under which the simulation is run calls `ltl2ba_start_monitor` when the simulation starts, and `ltl2ba_finish_monitor` when it ends.

There are a number of differences between our new approach and the old, the two most important being that the interleaving of the monitor thread with the program under test is explicitly encoded (see below), and the result of the BA applied to the trace is evaluated using the results of the static reachability analysis, rather than by only stutter extending the final state of the program under test.

Use of static analysis The output of the static analysis as described in Section 3.2.4.1 is encoded in the 3 arrays on lines 50, 54, and 57 of Listing A.2. Each state in the BA is numbered, as well as each input letter to the BA. The first array records whether a given state and input letter leads to an accepting infinite trace if stutter extended, while the other two arrays record whether a given state is only ever part of a good prefix, and whether a state can only be part of a bad prefix.

ESBMC can only report successful and unsuccessful verifications, and so we report the value in $\mathbb{B}_4$ of traces through in-program assertions. The `ltl2ba_finish_monitor` function performs this on line 67. Here, there are three assertions, corresponding to the $\top^p$, $\perp^p$, and $\perp$ results, which are violated when they identify one of those traces. $\perp$ is identified by the final state being one that can only be part of a bad trace; $\perp^p$ is identified by being in a state that accepts upon stutter extension; and $\top^p$ by the discovery of a trace that the BA does not reject. $\top$ is reported when the BA rejects for all traces[7] by ESBMC reporting a successful verification, or the BA terminates in a state that must be part of a good prefix.

As mentioned in Section 3.2.4.3, these conditions are not independent and so we check them in order, to find the least value of all traces through the program under test. As a result, our test harnesses run each benchmark 3 times, checking for only one response each time, in the order $\perp$ $\perp^p$ $\top^p$. The first of these values to be reported as a trace in the program under test is the least value, while if all three are not present then we conclude that the response is $\top$.

Synchronous Interleaving One of the flaws with our previous work was the composition of the monitor thread with the program under analysis in the same manner as we would any other thread, leading to unnecessary state explosion. We have therefore changed our approach to perform a deterministic and directed interleaving of the monitor with the program under analysis. Code inserted after global variable updates now calls a model checker intrinsic that causes it to context switch to the monitor thread, then context switch back once the monitor has

[7]No further transitions being feasible causes the transition `assume` statements to all evaluate to false, preventing any assertion from being satisfiable

run the BA a single step; the monitor itself no longer behaves as a schedulable thread. This technique effectively inlines the running of the BA at every point of interest. It also ensures that verification of single-threaded programs does not suffer from a multi-threaded state explosion.

3.2.7 Case Studies

We have tested the approach described in this paper against a set of behavioural properties of the pulse oximeter firmware (as introduced in the evaluation of our previous work, see Section 3.1.3), and a bicycle monitoring computer, a multi-threaded model of a data collection computer for cyclists. All tests were run on an otherwise idle Linux workstation [8] using ESBMC version 1.20[9] and Microsoft Z3 version 2.19, with a time limit of one hour to execute.

3.2.7.1 Pulse Oximeter

Here we report the results of verifying the pulse oximeter code against six properties selected from the SMV model of the software [69], as shown in Table 3.3. Note that all six properties hold for the code.

The source code to the pulse oximiter is listed in Section A.2, with our test harnesses and any modifications made documented in Section A.3.

Name	Property
baud_conf	$\mathsf{G}(\{\texttt{brate == 1200}\} \implies \mathsf{F}\{\texttt{TH1 == 0xE8}\})$
keyb_start	$\mathsf{G}(\{\texttt{the\_key == 1}\} \implies \mathsf{F}\{\texttt{command == 1}\})$
serial_rx	$\mathsf{G}((\{\texttt{p\_inDat == 1}\} \vee \{\texttt{flag2 == 1}\}) \implies \mathsf{F}\{\texttt{flag1 == 1}\})$
up_btn	$\mathsf{G}((\{\texttt{press == 4}\} \wedge \{\texttt{mstate == 1}\}) \implies \mathsf{F}\{\texttt{stime > refstime}\})$
start_btn	$\mathsf{G}((\neg\{\texttt{press == 1}\} \wedge \mathsf{F}\{\texttt{press == 1}\}) \implies \mathsf{F}\{\texttt{q\_startCall}\})$
buflim	$\mathsf{G}(\{\texttt{buffer\_size != 0}\} \implies \{\texttt{next < buffer\_size}\})$

TABLE 3.3: Properties for verification of pulse oximeter firmware.

The first four properties are liveness properties of the general form $\mathsf{G}(\varphi \implies \mathsf{F}\psi)$, so that whenever an enabling condition φ has become true, then eventually the property ψ is required to become true as well. These are some of the same properties verified in our previous experiments. The *up_btn* formula checks that when the up button is pressed `(press == 4)` and the device is in a particular state `(mstate == 1)`, then eventually an internal counter `stime` becomes larger than its previous value (kept in the variable `refstime` inside the test harness). The formula *start_btn* checks that whenever there is a transition of `press` to one from any other value then `q_startCall` will also become true now or in the future. Note however that we are not checking for a strict correspondence between changes in `press` and the occurrences of `q_startCall` becoming true so that for example the former can happen several times before

[8] 2.67Ghz Intel Xeon, 12Gb of memory, running Fedora 16

[9] Available from `www.esbmc.org`

			original		modified	
Property	*loc*	k	t (sec.)	Result	t (sec.)	Result
baud_conf	178	1	1	$\top^p$	1	$\bot^p$
		4	1	$\top^p$	1	$\bot^p$
		10	1	$\top^p$	1	$\bot^p$
		20	2	$\top^p$	2	$\bot^p$
keyb_start	50	1	1	$\top^p$	1	$\bot^p$
		4	2	$\top^p$	2	$\bot^p$
		10	5	$\top^p$	4	$\bot^p$
		20	17	$\top^p$	15	$\bot^p$
serial_rx	584	1	1	$\top^p$	1	$\bot^p$
		4	2	$\top^p$	2	$\bot^p$
		10	7	$\top^p$	5	$\bot^p$
		20	23	$\top^p$	25	$\bot^p$
up_btn	856	1	1	$\top^p$	1	$\top^p$
		4	1	$\top^p$	1	$\top^p$
		10	2	$\top^p$	2	$\top^p$
		20	3	$\top^p$	3	$\bot^p$
start_btn	856	1	1	$\top^p$	1	$\bot^p$
		4	1	$\top^p$	1	$\bot^p$
		10	2	$\top^p$	2	$\bot^p$
		20	3	$\top^p$	2	$\bot^p$
buflim	145	1	1	$\top^p$	1	$\bot$
		4	934	$\top^p$	4	$\bot$
		10	MO	MO	MO	MO
		20	MO	MO	MO	MO

TABLE 3.4: Results of testing LTL properties against pulse oximeter firmware.

the latter happens. Finally *buflim* is a safety property that ensures that a ring-buffer output index does not exceed the allowed limits. This check is similar to the buffer overflow checks already supported by ESBMC.

As before, we formulated test harnesses for each piece of code, and tested our approach with the original source code, and a version altered to be incorrect. The results are summarised in Table 3.4. Here, the *loc* column contains the line count of the source file for the portion of firmware being tested and k the loop unwinding bound specified for the test. The columns t and Result record the elapsed time in seconds that the test took to run and the outcome ESBMC reported for the test. A result of “TO” indicates the test did not complete in the allowed time, and “MO” indicates that ESBMC exhausted the available memory.

We first observe that ESBMC determines the expected result for most test cases. Since the first five properties are liveness properties, ESBMC reports the inconclusive results $\top^p$ and $\bot^p$ instead of the definitive values. We also observe that the amount of time taken scales roughly linearly with the unwind bound given in most tests. A notable exception is the *buflim* test, which increases dramatically in time and memory requirements. This performance hit is caused by a large amount of program non-determinism in the portion of code being LTL checked, making the checking of higher unwind bounds unfeasible.

Finally, we observe that the *up_btn* property has incorrect results for a number of cases. Here, the seeded error combines a number of (in this case, three) consecutive keypresses into one keypress event. This violates the property that the internal counter `stime` always increases after the enabling key press event. However, as every third keypress *does* result in a keypress event, the unwind bounds of 1, 4 and 10 terminate with the most recent keypress having caused a keypress event, thus terminating in a $\top^p$ state. This is an example of a property that oscillates between $\bot^p$ and $\top^p$ as the unwind bounds are changed, as discussed in Section 3.2.5.3.

3.2.7.2 Bicycle computer

The bicycle computer case study comprises a small C-model of a device designed to gather and display speed and distance information about a cyclist's journey. This case study contains approximately 150 lines of code. The program is multi-threaded and treats user input, display, and data collection as separate processes. We test a number of (valid) properties over the global state of the program, listed in Table 3.5. The source code to the bicycle computer is listed in Section A.4.

Name	Property
dist_ovfl	$\mathsf{G}(\{$`cycle_distance_m >= 0`$\})$
tot_dist_ovfl	$\mathsf{G}(\{$`total_cycle_distance_m >= 0`$\})$
dist_rel	$\mathsf{G}(\{$`cycle_distance_m <= total_cycle_distance_m`$\})$
state_range	$\mathsf{G}(\{$`cur_state >= 0`$\} \wedge \{$`cur_state <= 3`$\})$

TABLE 3.5: Bicycle computer properties.

		$k=1$		$k=2$	
Property	C	Time (s)	Result	Time (s)	Result
dist_ovfl	1	1	$\top^p$	1	$\top^p$
	2	7	$\top^p$	34	$\top^p$
	3	56	$\top^p$	379	$\top^p$
tot_dist_ovfl	1	1	$\top^p$	1	$\top^p$
	2	5	$\top^p$	24	$\top^p$
	3	63	$\top^p$	368	$\top^p$
dist_rel	1	1	$\top^p$	2	$\top^p$
	2	7	$\top^p$	32	$\top^p$
	3	59	$\top^p$	542	$\top^p$
state_range	1	1	$\top^p$	2	$\top^p$
	2	7	$\top^p$	42	$\top^p$
	3	62	$\top^p$	478	$\top^p$

TABLE 3.6: Results of testing LTL properties against bicycle model.

Because this program is multi-threaded, checking it using ESBMC results in a large number of distinct runs of ESBMC's SMT solver, each corresponding to different thread interleavings. These have to be combined together to report the worst (in the four-valued lattice) behaviour of any interleaving.

We test the program against the properties with a number of different unwind bounds k and context switch bounds C. Our results (cf. Table 3.6) show the correct output is determined for each run, for a variety of loop unwind bounds and context switch bounds. We note that verification time increases exponentially with increases in the context bound, which is as expected in multi-threaded verification.

The bicycle computer examples above are all safety properties. Verification of liveness properties in multi-threaded code presents additional difficulties for our approach and is currently practical only for small examples. Multi-threaded safety failures are typically *shallow*, requiring only few interleaves. In contrast, even liveness properties guaranteed by loop invariants require that relatively large interleave bounds be set to ensure that all threads run complete loop iterations. More general liveness properties can depend on scheduling between threads. The default pthread behaviour provides weak fairness and is accurately modelled by ESBMC. Liveness properties which depend on having fairness will, however, inevitably show violations for finite traces, as ESBMC will produce a thread schedule where the unwinding bound of a thread that makes a response to a request is exhausted prior to a request being made.

3.2.8 Analysis

The most obvious difference from our previous work is that the amount of time required to evaluate a formula over the system under test is vastly reduced, for the pulse oximeter down from an average of thousands of seconds in Table 3.1, to tens in Table 3.4. As a result, the unwinding bound that we are feasibly able to reach is greatly extended, from one or two up to twenty. This improvement is entirely due to the change from using multithreaded exploration to find the synchronous interleaving, to explicitly exploring it. (It also eliminates for single threaded programs the relevance of the context switch bound, which is not displayed in Table 3.4).

We also receive more meaningful responses from the verification process, with the liveness properties evaluating to $\top^p$ and $\perp^p$ rather than delivering a definitive result, and the safety properties reporting $\top^p$ and $\perp$ for successful and failing verifications. We are also able to use the BA static analysis on the formulae we verify to determine what results to expect—as a result, we know that no formula is a co-safety property and thus none can yield a $\top$ response, and so we are not surprised that it is not reported in any of our tests.

We show that our approach applies correctly to multithreaded programs as well as single threaded, with the caveat that weak fairness of thread interleavings interferes with the evaluation of liveness properties.

3.3 RERS Greybox challenge

In order to evaluate our approach against a larger set of benchmarks, and to compare it against other methods of verify LTL properties, we entered ESBMC into the 2012 RERS Greybox Challenge [149], a competition to verify LTL and reachability properties across a suite of synthesised benchmarks classified as "reactive systems".

In this section, we introduce the benchmarks that make up the RERS challenge, the properties, our approach to verifying them, and our results. We then compare the performance of our approach against other tools submitted to the challenge. The work here was originally published in [127].

3.3.1 Introduction

The premise for the RERS challenge is that there are insufficient test suites available for uniformly evaluating software verification tools—industrial samples tend to have many environmental dependencies, and focus on very specific scenarios [149]. They also do not allow for verifiers with differing input languages to be directly compared. To address this, the organisers developed a method of generalising forms of behavioural properties (expressed in LTL), combine a set of them into an abstract state machine, then synthesising benchmarks representing those state machines into several different languages for verification by different tools.

Prior to the conference, the organisers published these synthesised benchmarks and sets of properties for each, and set the task of analysing the benchmarks to determine which of the properties hold over the systems. The correct results were not published in advance, and the competitors submitted their results before the reference answers were released. The term "greybox" refers to the nature of the testing being performed: the competitors can study the structure of the system under test directly so it is not a "blackbox" test, but there is no meaning associated with the synthesised test's actions, preventing it from being a "whitebox" test.

The properties came in two flavours: firstly, the LTL properties were a mixture of safety, co-safety and liveness properties. Secondly, reachability properties were present, however they are not within the scope of this chapter and will be ignored for the remainder (see the publication [127] for more information).

As well as the pre-published benchmarks, the organisers also published another set of benchmarks during the conference at which competitors results were discussed, with the intention that they be analysed during the course of the conference. This was referred to as the "online" challenge, the other benchmarks "offline"; we did not participate in the online challenge.

3.3.2 Challenge problems

Briefly, the test suite of benchmarks were generated thus: a set of LTL formulae were generated from several formula templates, expressed over abstract states. These formulae were each converted into Büchi automata, which were then combined into a single Mealy state machine. The transformation process ensured that all the traces accepted by the BA are also accepted by the Mealy machine. This machine was then synthesised into software implementations in Java and C. In these implementations the machine states are not given explicitly, but only implicitly by the possible values of a number of state variables. The implementations consist of a main loop which in each iteration reads an input (i.e., event) from the standard input, updates the state variables, and possibly writes an output (i.e., action) to the standard output; the latter two are guarded by conditionals over the input, and over the values of the state variables.

The offline challenge consisted of nine benchmarks (labelled `Program1` to `Program9`), each with 100 LTL formula to be checked, and were classified on two dimensions of *size* and *complexity*: for size, benchmarks were labelled one of "small", "medium", or "large", and for complexity the labels were "easy", "moderate" and "hard". "Larger" benchmarks had more state transitions within them than the smaller ones. The exact meaning of the differing levels of complexity was unclear, although all three hard problems had substantially more state variables.

The challenge problems all work with relatively small alphabets, and use five or (in most cases) six different input symbols, and between three and nine different output symbols. Easy and moderate problems have between four and eight state variables, while large problems have thirty. The programs for the offline problems only assign up to five different integer constants to the state variables, and only use the equality and propositional operators in the guards.

3.3.3 Execution of Experiments

We participated only in the offline phase of the Challenge, and attempted only the small and medium problems (i.e., `Problem1` to `Problem6`), of which we only obtained results for LTL properties from the first four. The large problems (`Problem7` to `Problem9`) were too large for ESBMC's analysis, and lead to memory exhaustion during parsing. We ran ESBMC on the C versions of the Challenge programs, with some minor modifications: replacing input from `scanf` by an appropriately constrained non-deterministic choice, and to pruning (by means of an assumption on the computed output) executions that use invalid inputs. The input and output values were also promoted to global variables so that we could make expressions in the LTL formula about their values.

We ran all the experiments on the Southampton IRIDIS compute cluster,which comprises about 1000 nodes, each with 12 2.4Ghz Intel Westmere cores and 22Gb of memory, running Red Hat Enterprise Linux Server release 5.3 (Tikanga). We submitted batches of 60 jobs, which where scheduled by IRIDIS' own job scheduling system. We set no time or memory limits for the jobs corresponding to the reachability properties, and a time limit of one hour (but no memory limit) for the jobs corresponding to the behavioural properties. Each property was checked with an increasing number of unwinding bounds, initially from five to twenty; the harder tests, however, mostly became unfeasible to verify after fifteen unwinds. The exact responses received for each unwinding are in Table 3.8.

We did achieve some responses for `Problem5`, however all of the reachability properties that we checked at the same time evaluated as unreachable, which seemed suspicious. As a result we reasoned that some bug within ESBMC was making some paths unfeasible, and chose not to submit results for `Problem5`.[10] We managed to verify some reachability properties in `Problem6` at high unwind bounds, but not the LTL properties, and chose not to report the LTL properties for that problem.

3.3.4 Approach

The LTL formula given by the organisers required a small amount of transformation before being suitable for our approach: a *weak-until* operator was replaced with $\mathsf{G}\phi \bigvee \phi \mathsf{U} \psi$, and propositions about the input letter to the state machine were replaced with C expressions inspecting program input variables. The approach we have developed (as described above) then applies cleanly, and we produce BAs and monitors as explained in Section 3.2.

For verifying the LTL properties, we also ignored certain illegal paths through the benchmarks (signified by assertion failures) by transforming calls to the `assert` function to `assume`. This had the effect of pruning all illegal paths from being were explored by the model checker.

It is important to note at this point that the only actions taken to make our LTL verification approach apply to these benchmarks were to expand the weak-until operator, and to make the input and output values global variables. The other changes were to correctly model the nondeterministic behaviour of the benchmarks within our model checker, which was also required to verify the RERS reachability properties. The weak-until operator is not part of the usual LTL definition and so we are justified in having to transform it; the limitations of requiring expressions over global variables are discussed in Section 3.4.1.

For the competition itself, competitors were invited to identify each LTL property for each benchmark as either holding or having a violation, and also to assign a confidence to each individual result on a scale of one to nine. A score was the computed, with the weight of the confidence accumulated if the property was correctly classified, or doubled and subtracted if it was not. We assigned a confidence of nine to $\top$ and $\bot$ responses, as in our approach these definitively identify properties holding and being violated. Our choice of confidence for $\top^p$ and $\bot^p$ responses was guided by the stability of the response as we increased the unwinding bound, as discussed in Section 3.3.6

[10] A decision vindicated as a later version of ESBMC, 1.23, is able to identify reachable properties in `Problem5`.

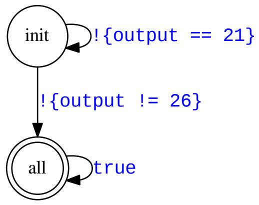

FIGURE 3.5: The BA generated for the never claim of the property *output U occurs before output Z.*

3.3.5 An example

We take as an example the LTL formula for the first behavioural property for the small/easy case, i.e., the output U occurs before output Z:

```
(! oZ WU (oU & ! oZ))
```

After translation into our input format, the never claim becomes

```
!(({output != 26} U ({output == 21} && {output != 26}))
  || (G {output != 26}))
```

The BA for this formula is shown in Fig. 3.5. This particular LTL formula does not fall into any of the three simple types of property, safety, co-safety, or liveness. A finite prefix[11] can be good (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oU} \rangle$, where the BA fails) or bad (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oZ} \rangle$, where the BA is guaranteed to be able to remain in an accepting loop). It can also be succeeding (e.g. $\langle \texttt{oV}, \texttt{oV}, \texttt{oV}, \texttt{oV} \rangle$, where both success and failure remain possible but an infinite stutter extension would be good). This particular BA cannot, however, show failing behaviour.

3.3.6 Analysis results

We were only able to achieve useful unwind bounds on the three small problems and the medium/easy problem (i.e., `Problem1` to `Problem4`). Table 3.8 summarises the results. For all small problems, all outcomes are the same for unwind bounds 9–14. We thus have reasonable confidence in our results for the small problems.

For the medium/easy problem there are a few properties (#0, #14, #17, #77, #98) where the outcome changes with increasing unwind bounds. However, in all cases the change is from $\perp^p$ to $\top$, corresponding to finally reaching the co-safety witness with the next iteration of the program's loop.

[11] Since this specific LTL formula only uses `output` the traces (and thus prefixes) consist of `output`-literals only. However, the corresponding `input` values can still be extracted from the BMC counterexamples.

Overall, the definitive (i.e., good and bad) and inconclusive (i.e., succeeding and failing) outcomes are roughly equally common. However, we find substantially more decisive results (200 instances of $\perp$) than probable results (12 instances of $\perp^p$)

We used a validation program (`Problem10.c`) to validate our analysis results—this problem was provided with reference correct answers to allow competitors to check their approaches. For the 100 given LTL properties, our approach produced, with the scheme outlined above, only two false results (for #13 and #30). In both cases, we claim that the formula is succeeding, while the validation suite claims an explicit counterexample. However, in both cases the counterexample involves invalid inputs, which we have explicitly ruled out.

We thus submitted every $\top$ ($\perp$) case as the property holding (not holding). These results were given a weighting of 9, the strongest, showing complete confidence that the result is correct. since we get explicit witnesses (counterexamples). The succeeding and failing cases are more problematic; based on the results we achieved over the validation suite, we have chosen to report them, even for the medium/easy code, as success and failure with weightings of 7 and 9 respectively. We weight the failing cases with the greatest confidence as they occur in execution traces where a partial violation has been found, however for the succeeding traces we fear that a violation may yet occur if the unwinding bound were extended further.

3.3.6.1 Discussion

For the 400 properties we analysed we returned 385 (96.3%) correct results, which gives us, with the weights as explained above, a total score of 2991 marks. This compares fairly well to the results achieved by the other teams taking part who for the same set of benchmarks achieved the results in Table 3.7.

Team	Score	Total answers	Incorrect answers	Percent correct
Twente	3492	400	4	99.0%
Paris	3069	362	7	98.1%
Southampton	2991	400	15	96.3%
Vienna	1665	248	21	91.5%

TABLE 3.7: Scores and results for other RERS'12 participants who returned results for LTL properties, on programs one to four, retrieved from [150]

The 15 wrong results fall into two different categories. In five cases, we find that the program is failing (succeeding) with regard to the property, but the failure (success) result that we report is wrong, because our unwind bounds are too small. In the remaining cases we find that the program is bad with regard to the property, but the counterexample trace goes through an error state; this trace should eventually be pruned away (using an `assume(0)`-statement) at an error label, but the automaton accepts a number of additional inputs sufficient to push this error label over the unwinding bound.

In terms of overall score however, our entry came last as we were unable to submit results for the majority of the tests. This is mostly because of the performance bounds reached when evaluating the larger and harder tests, rather than any limitations of our LTL verification approach.

However, it seems to be clear that symbolic bounded software model checking is not the optimal technique for the RERS competition: the programs implement finite state machines with a relatively small state space, but bounding and unrolling under-approximates the reachable state space while at the same time the symbolic valuation of state variables over-approximates it. Similarly, the programs are much simpler than those typically encountered in software model checking (e.g., the offline problems only use integer equality and contain no other operations or data structures) while at the same time the large programs (approximately 70,000 to 180,000 lines of code) are too large to unroll them sufficiently often.

3.3.7 Conclusions

Clearly, symbolic bounded model checking is not the best tool for solving the problems in the RERS challenge—we could not handle many of the larger tests, and for the reachability properties at least, ESBMC is orders of magnitude slower than Java Pathfinder, an explicit-state model checker for Java [156].

However, we are encouraged that ESBMC, a general-purpose multi-threaded C model checker, has been able to generate useful analyses of these large and somewhat unusual systems. For the LTL properties, we produced 15 wrong results and achieved a success rate of 96.3%, which is relatively close to the winner's success rate of 99.0%. We believe that our software model checking approach will become more competitive as the programs become more complicated (e.g., use of larger alphabets, arithmetic operations in the state updates, or data structures), and plan to participate in future Challenges with such problems.

In fact, revising the benchmarks used in RERS nearly two years later (in preparation for the submission of [127]), we discovered that the latest developmental version of ESBMC evaluates all the properties two orders of magnitude faster than when we originally took part, increasing to three orders when using a faster SMT solver (see Chapter 4). This greatly increases the number of unwind bounds that we can feasibly reach.

3.4 Discussion

In this chapter, we have taken an existing approach for verifying LTL properties over a model of a system under test, and have extended it into a form where it can be directly applied to an ANSI-C software code base, without modification, permanent transformation, or significant technical expertise from the verification engineer.

We have evaluated our approach, and made improvements where necessary to increase its performance to a level where it can verify LTL properties reasonably quickly (i.e. in less than an hour).

The method has been tested on a series of benchmarks and properties that are both of our own devising and also taken from an industrial piece of software (the pulse oximeter). We have also used our method to verify synthesised properties in a verification competition, have have competed reasonably well (in terms of accuracy) against the other participants in the competition.

We have also explored how the value that an LTL property evaluates to can be extended to provide more meaningful information about the manner in which the property holds over the system under test, and used it to verify liveness properties in ANSI-C software.

The precise contributions of this chapter are:

- The first application of LTL model checking to an ANSI-C code base.
- The first model checking method for full LTL that uses bounded model checking, rather than BDD's.
- This first bounded model checking method that gives meaningful information about liveness properties.

3.4.1 Limitations

Our approach does operate under a few constraints, which we enumerate here.

3.4.1.1 Bounding of loops

The premise of our approach is that we are dealing with finite prefixes of infinite traces. This allows us to bound the outermost loop in the program, and the bound determines the length of the prefix explored. Bounding inner loops, however, could cause their early termination resulting in a trace that is not a true finite prefix on an infinite trace. With a small amount of work we could independently bound them and apply an unwinding assumption (detailed in Section C.1.3) that ensures the only loop exits permitted do so in a consistent state. This however would reduce the completeness of the verification, as traces with deeply unwound internal loops would not be examined.

In practice none of the systems that we have verified have had such inner loops. Additionally, the embedded systems that ESBMC targets often implement some form of state machine or have a single outer event loop around all their behaviours.

3.4.1.2 Liveness properties in multithreaded systems

As discovered in Section 3.2.7.2, the verification of liveness properties against multithreaded systems is still difficult as the weak fairness that ESBMC's thread scheduler models will inevitably produce a schedule that evaluates to $\perp^p$. This is a limitation caused by the nature of multithreaded model checkers however, and not by our approach to LTL model checking.

3.4.1.3 Global variables

Our approach requires that the expressions in the LTL formulae only refer to global variables in the program under test. This is partially to avoid having to specify the scope in which each C expression should be evaluated, but more generally because function-locally scoped and dynamically allocated variables have a lifetime independent of the program under test, whereas global variables exist and have a defined value for as long as the program is running.

Were such non-global variables to be addressable in the LTL formula we verify, the semantics of the lifetime of the variable may need to be defined: for example, does the property hold when the variable is out of scope, does the property become violated when the variable ends its lifetime? Recursion leading to multiple instances of a lexical variable becomes particularly gnarly.

Again, the embedded software that ESBMC tends to be applied to have many global state variables and few local variables as stack depth on microcontrollers is limited. They also tend to have no dynamically allocated memory at all.

3.4.1.4 Expression Independence

The ANSI-C expressions used in our LTL formula may not be independent—for example, $\{\texttt{a == 2}\}$ and $\{\texttt{a == 3}\}$ cannot be true at the same time. Our static analysis of the automata is unable to detect this, and as a result may misclassify formula. Consider $\{\texttt{p}\} \implies \mathsf{F}(\{\texttt{q}\} \wedge \{\texttt{!q}\})$: if `p` ever becomes true then the future condition can never be satisfied, meaning any state where `p` holds is a bad prefix. However, as the analysis algorithm is unaware that the two C expressions `q` and `!q` are not independent the worst result we find is $\perp^p$ [12]

This flaw does not seriously affect the result of our approach as it can only make the responses more conservative, by considering that unfeasible transitions might occur. It will only ever weaken $\top$ or $\perp$ to their $\top^p$ and $\perp^p$, respectively. Significant effort would be required to analyse the C expressions and identify common ANSI-C variables in them, and even then different C variables may be dependant due to actions in the program under test; identifying such behaviour is a model checking problem in itself.

[12] Note that if we had written $\neg\{\texttt{!q}\}$ then the ltl2ba algorithm would have been able to identify the accepting trap. The ANSI-C expressions are identified as being the same proposition if the parse tree of the expression is identical

3.4.1.5 Indirect variable accesses

As identified in Section 3.1.2.2, our approach does not currently handle indirect assignments to lexical variables. The solution to this is however trivial and explained in that section.

3.4.2 Future work

The most beneficial effort in the future to improve our method, would be identifying some strong-fairness guarantee that can be supplied in ESBMC's multithreaded model checking behaviour, that would allow for liveness properties to be verified. As it stands, this is the only serious limitation of formula that can be checked from which our method suffers. Such a guarantee is fraught with difficulties though, because in symbolic execution it is difficult to ensure that one thread "runs as often as all other threads", as there is no good definition of how much progress is made by the execution of one thread. Neither assignments executed nor number of visible statements reasonably map onto "time elapsed", on which fairness is defined.

Otherwise, an automated method of evaluating a program as the unwinding bound is increased would be beneficial, as our method defines a "well behaved" liveness property as one where $\top^p$ is reported frequently as the bound is changed. Recent developments [88] may ease this process.

Improved performance of this method would be of use, in terms of verification time and the size of input that it is able to interpret. Such improvements would have to be entirely provided by the underlying model checker (ESBMC), and are perennial demands on model checking tools.

3.4.3 Related work

We cover the background of LTL model checking in Section 2.3.3. Here we examine existing LTL model checkers that are comparable to our approach.

SPIN [92] is a well known software model checker that operates on concurrent program models written in the Promela modelling language. SPIN operates with explicit state and uses state hashing to reduce the quantity of state space it explores. SPIN also allows users to specify an LTL formula to verify against the execution of a model by using BA in a similar manner to our work. While SPIN is well established as a model checker, the requirement to re-model codebases in Promela can be time consuming.

Java PathFinder is a Java Virtual Machine (JVM) that performs model checking on Java bytecode. It also operates with explicit state and uses *state matching* to reduce the search space, but can also operate symbolically for the purpose of test generation and coverage testing. Verification of LTL formulae can be achieved with the JPF-LTL [132] extension which uses BA and method invocation monitoring to inspect the execution of the model.

Staats and Heimdahl [148] take Simulink models and verify that a prototype Simulink-to-C translator produces code that satisfies the same properties as the Simulink model. A set of predetermined safety properties described in LTL are verified first against the Simulink model, then against the emitted C code. A C monitor is devised, and a feature of the converted model is used to select code locations where the monitor must be inserted. Their approach is not designed to support the checking of liveness or co-safety properties.

Leucker and Schallhart [116] review the field of *run-time verification* and cover its differences from model checking, as well as various LTL-like logics for analysing finite prefixes of traces. More expressive ways of describing system properties are explored, as well as the potential for run-time analysis beyond verification.

The DIVINE model checker [21] is an explicit state model checker that supports full LTL specifications over finite state models. Their recent work [19, 142] has focused on supporting LTL model checking in C and C++ software, through the use of the LLVM compiler infrastructure. Their work differs from ours in two significant aspects: first, as an explicit state model checker they do not deal with finite prefixes of an infinite trace, but instead attempt to find an accepting loop through a finite model by enumerating the full state space [20]. Second, they annotate the source file with (sometimes conditional) directives to set the propositions that the LTL formulae are expressed over.

3.4.4 Conclusions

Context-bounded model checking has already been used successfully to verify multi-threaded applications written in low-level languages such as C. However, the approach has largely been confined to the verification of safety properties. In this paper, we have extended the approach to the verification of liveness properties given as LTL formulae against an unmodified code base. We follow the usual approach of composing the BA for the never claim with the program, but work at the actual code level. We thus convert the BA further into a separate C monitor thread and check all interleavings between this monitor and the program using ESBMC. We use a four-valued LTL semantics to handle the finite traces that bounded model checking explores.

Our results so far are encouraging, and we were able to verify a number of liveness properties on the firmware of a medical device; in future work, we plan to extend the evaluation to a larger code base and wider variety of properties. There are still considerable opportunities to improve performance and to execute on more capable computer platforms. For multi-threaded simulations, the state hashing reported in our SEFM 2011 contribution [125] has proved to be very useful, cutting verification times by about 50% on average. We expect that an improved hashing implementation, for example removing serialisation, will improve these results further.

	Problem1						Problem2						Problem3						Problem4					
$\backslash n$	9	10	11	12	13	14	9	10	11	12	13	14	9	10	11	12	13	14	9	10	11	12	13	14
0	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
1	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–	–
2	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
3	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
4	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
5	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
6	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
7	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
8	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
9	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
10	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
11	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
12	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
13	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
14	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
15	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
16	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
17	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\boldsymbol{\top}$
18	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
19	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
20	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
21	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
22	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
23	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	–	$\top^p$	–
24	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
25	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
26	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
27	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
28	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
29	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
30	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
31	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	–	–	–
32	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
33	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	–	–	–
34	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
35	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
36	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
37	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
38	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
39	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
40	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
41	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
42	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
43	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
44	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
45	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
46	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	–	–	–
47	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
48	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
49	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
50	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
51	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
52	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	–	–	–
53	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
54	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
55	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
56	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
57	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
58	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
59	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–	–
60	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
61	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
62	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
63	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
64	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–
65	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
66	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
67	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
68	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
69	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
70	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
71	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
72	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
73	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
74	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
75	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
76	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	–	–	–	–
77	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	$\boldsymbol{\top}$
78	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
79	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	–	–
80	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
81	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
82	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
83	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–
84	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
85	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
86	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
87	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
88	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
89	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–
90	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
91	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
92	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
93	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
94	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
95	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	–	–
96	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	–	$\top^p$	–
97	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–
98	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot^p$	$\boldsymbol{\top}$	$\top$	$\top$	$\top$	$\top$
99	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	–	–

TABLE 3.8: Results for behavioural properties. Only claims are shown, for different unwinding bounds ($n = 9$ to $n = 14$). "–" denotes a time-out ($t_{\max} = 3600s$). Boldface denotes changes in the outcomes as the unwinding bounds change.

Chapter 4

Efficient solvers and encoding for SMT formulae

Many symbolic model checkers operate by transforming their input into a form suitable for an automated theorem prover to process, separating the concerns of producing a formulation of the model checking problem, and actually exploring whether the formula is true or not. In the case of ESBMC and similar solvers, this corresponds to the way in which high level concepts of the C language are translated into quantifier free first order logic,[1] to be solved by an SMT solver.

SMT and SAT solvers are popular with software model checking, partly because bounded model checking originated through the use of SAT solvers [37], but also because the input format correlates closely with how imperative software operates. Symbolically represented variables with constraints can easily represent the computations performed during a program execution, and SMT solvers support expressions over integers, reals, bitvectors and arrays which matches the operation of imperative software operation. SMT itself is brought together by the *SMTLIB* standard [24], which defines a syntax and semantics for describing SMT formulae, the features that solvers must support, and how the solver is to be configured. The standard also defines a set of logics, each of which defines a set of features that may be used in formulae. For example, "BV" allows the use of quantifiers and bitvectors, while "QF_AUFLIRA" disallows quantifiers, and allows arrays, uninterpreted functions, and linear arithmetic of integers and reals.

The manner in which model checkers solve reachability problems with such solvers is of great interest, as it governs the performance with which model checking problems can be solved. Details on precisely how different model checkers implement their translation are, however, sparse. Convention in the research field [31] means that publications rarely descend to such a level, preferring to focus on high level formulations of the encoding and theoretical advances. I have been unable to find any technical reports that describe the way in which high level programming constructs are reduced to first order logic, except where that particular feature is the subject of

[1] i.e., propositional logic with theories of equality, optional theories such as bitvectors, arrays and linear arithmetic, but individual variables cannot be quantified. The whole formula is implicitly quantified over the value of all variables

some optimisation or other interest. In the case of CBMC, from which ESBMC inherits, the closest and most detailed document is [111], which focuses on the high level decomposition of C down to CBMCs internal execution format, and then on to assignments. LLBMC [121] too offers a high level view, mostly concerned with the decomposition of LLVM bitcode into an unrolled program.

I believe this situation is severely sub-optimal, as greater knowledge and understanding of how different tools perform their encoding can only lead to discussion and technological advancement. Radically different encodings of multithreaded verification problems (discussed in Section 5.6) have shown large performance improvements on ESBMCs approach; encouraging greater experimentation with such encodings would almost certainly advance the field. To aid this, I have described ESBMCs encoding in detail in Appendix C, which is relevant but not necessary reading for this chapter.

In this chapter I examine how to improve the performance of ESBMC by increasing the speed at which SMT formulae are solved. Two matters are considered: firstly, which SMT solver is the fastest to solve the particular class of formulae that ESBMC produces. Secondly, how ESBMCs encoding of an unrolled (SSA) program to SMT can be optimised for faster solving times.

I select the set of SMT solvers used by similar model checking tools to ESBMC and evaluate them, the widest such evaluation I am aware of in the literature. I identify two techniques for improving the construction of memory references which have not been evaluated before and compare them. I also evaluate the effectiveness of a previously proposed technique for reducing the size of the SMT formula encoded.

For evaluation, I use the International Competition on Software Verification [31, 32, 33] (SV-COMP) suite, from the 2014 competition. SV-COMP, being an aggregation of benchmark suites used by software model checkers (for the C language, the largest such suite collected to my knowledge), is suitable for verification as almost all benchmarks have a single well defined reachability property to check[2], and with 2877 tests from 12 sources [73] is sufficiently large and diverse to have confidence that a wide range of C language features are exhibited by the suite.

I do not consider the SV-COMP *score* achieved in this chapter. ESBMC reports the wrong result for some benchmarks (due to an insufficient unwinding bound, or bugs within ESBMC), and incorrect results are seriously penalised by the scoring scheme. This gives rise to scenarios where an optimisation increases the number of SV-COMP benchmarks that successfully complete and report a result, but a lower score is achieved because the newly reported results are incorrect. As we care here about improving performance and thus increasing the number of completed verifications, the SV-COMP score is irrelevant.

[2] The "memory safety" set of tests have implicit properties, that the memory accesses performed by the programs are always correct

Organisation Section 4.1 identifies different SMT solvers that can be used and tests ESBMC with them. Section 4.2 examines how memory accesses in ESBMC are encoded and experiments with different encoding approaches. Section 4.3 details a more efficient encoding of the execution guards used by ESBMC. Section 4.4 draws conclusions.

4.1 Extending solver support

Numerous SMT solvers are available that comply with the SMTLIB standard, and use a variety of techniques and heuristics to optimise the solving of formulae. The community also runs a performance competition [23] to evaluate which solvers are the best, categorised by the different logics that they support.

An obvious question to ask is which SMT solver is the fastest at solving the formulae produced by ESBMC? The current default is Z3 [72], as a previous evaluation with ESBMC [70] found it to be the fastest solver at the time. In the intervening five years however there have been substantial improvements in SMT solving technology, and additional solvers are now available.

Within ESBMC's peers at the Software Verification competition a variety of SMT solvers are used. Z3 is the most popular, and most of the participants of SMT-COMP are used by at least one model checker. Some tools make use of SAT and constraint solvers too. The literature does not appear to feature any robust comparison of SMT solver performance when applied to automated verification, aside from ESBMCs previous evaluation [70] and an evaluation for path feasibility in the context of symbolic execution [134].

To answer the question of performance, I select the best performing solvers from SMT-COMP'14 [65] for the logic that ESBMCs formulae are encoded in, QF_ABV (quantifier free with arrays and bitvectors), and that are also used by our peers at SV-COMP'14. These solvers are described in Section 4.1.1. I omit the solver SMTInterpol from this list as it operates in the Java Virtual Machine, which is incompatible with ESBMCs native environment and would require a large amount of time to work around.

Of the solvers described below, I would expect Boolector to perform the best, given that it wins the QF_ABV category of SMT-COMP'14. This would be in contrast to ESBMCs previous evaluation which found Z3 to be the fastest.

During this evaluation I also implemented support for the metaSMT framework [135], which offers a common interface to a suite of SMT solvers. I did not run any experiments with it however, for two reasons. First, the primary interface to metaSMT is via a domain specific language that is statically converted to an abstract syntax tree at compile time (through the use of Boost.Proto), which is not compatible with the dynamic creation of formula at runtime that ESBMC performs. Secondly, when directly interacting with metaSMT's "middle end" [135] to dynamically construct formula, significant overhead was introduced during conversion time, to the extent of frequently increasing runtime by 50%. Given these limitations, I chose instead to manually implement solver backends myself.

4.1.1 Supported solvers

The **Z3** solver [72], from Microsoft Research, is a general purpose theorem prover that accepts SMTLIB as an input. A wide range of input logics are permitted, and Z3 has numerous extensions to allow for complex datatypes (such as tuples), as well as lists, sets, and recursively defined datatypes. Z3 also possesses a fixedpoint engine, known as μZ3. Of all these features, ESBMC only uses the SMTLIB input and tuple support. Z3 has a history of scoring highly in SMT-COMP, and won more than half of the categories in 2011 [63]. It has not, however, entered the competition since, and has not had a release since January 2013, leading me to believe that Microsoft have ceased maintenance for the project. Z3 is distributed under the Microsoft Research License Agreement (MSR-LA), and has had its source code released under the same license since 2012.

Yices [77] is a SMT solver developed at SRI international, supporting all of the SMTLIB standard, as well as extensions such as MAX-SMT problems. Yices is used by SRI in internal projects, but is also distributed in binary form under a proprietary non-commercial license. Yices version 2 took part in SMT-COMP14 [65], and won 10 out of 34 of the categories.

Boolector [44] is a SMT-flattening tool that takes formulae written in SMTLIB (or Boolectors own input format) and converts them to SAT formulae. Several SAT solvers can be chosen to solve the subsequent formula. Boolector focuses on producing an initial abstraction of the SMT formula, then refining it to be more accurate as the SAT solver generates satisfying assignments. Boolector has been released under academic-free and GPL3 licenses in the past, and since the latest version (1.6) is released under an MIT-like license with non-commercial and no-competition-use clauses. Of the two categories that Boolector entered in SMT-COMP14, it won both.

MathSAT [51] is a general purpose SMT solver, with aims to be a fully featured and long lived solver, for use in academic as well as industrial contexts. MathSAT supports all the usual SMT logics, but also supports the creation of Craig-interpolants (see Section 2.1.5.3), partial assignment enumeration, and other features requested by industry. MathSAT's license is non-commercial academic-free. It last competed in SMT-COMP in 2012[64], when it won one category and entered three more.

CVC4 [2] is a theorem prover with SMTLIB compatibility, which supports a wide range of logics, including those with quantifiers. CVC is a collaboration between many US universities, and has a long and detailed history. It is distributed under the terms of the new BSD license. It took part in SMT-COMP14, winning 7 out of 34 categories, and participated in all but two of the available categories.

As well as backends for interacting with the above solvers, I have also implemented a backend that works with the textual interface to SMTLIB solvers. ESBMC and any SMTLIB compliant solver can communicate across a socket or pipe using SMTLIB syntax, allowing any additional solver for which ESBMC does not have a backend to be plugged in, albeit with an overhead due to serialising information through a communication channel.

4.1.2 Comparison

I now examine the performance of each SMT solver, when applied to the SV-COMP'14 [33] benchmark suite. ESBMC was run on each benchmark in the suite once per solver, with the following command line:

```
esbmc --unwind 8 --no-unwinding-assertions --timeout 15m
--memlimit 15g --64 --tuple-sym-flattener --no-slice
--context-switch 3 -DLDV_ERROR=ERROR -Dassert=notassert
-D_Bool=int --no-assertions --no-bounds-check
--no-pointer-check --error-label ERROR
--no-div-by-zero-check
```

This enforces the operational constraints of SV-COMP, limiting ESBMC to consuming only 15 minutes of runtime, and 15GB of memory. Other options configure ESBMC to only report reaching the label `ERROR` as a verification failure, with all other program properties being disabled. A number of performance flags are set (such as `--no-slice`), and workarounds for compiling certain benchmarks (such as defining the `_Bool` symbol as an integer).

The unwind bound is set to 8 loops, as this is the highest bound we have used when submitting ESBMC to SV-COMP in the past, and has given us the greatest score. The multithreaded context bound is set to 3 context switches for the same reasons.

In this section, I compare the performance of different solvers by looking at the number of benchmarks that either crash in out-of-memory conditions or timeout when ESBMC uses that solver, and the amount of time consumed across all the benchmarks that did not crash or time out under any solver. In addition, I omit 17 tests where ESBMC could not parse the input file due to faults in the C parser, which does not support floating point hexidecimal constants or the C11 _Thread_local keyword. The omitted tests are presented in Table 4.1.

The version numbers of the solvers used in this evaluation are presented in Table 4.2. I deliberately chose an older version of Z3 (4.0 versus 4.3) as Z3 releases from 4.1 onwards have a serious performance regression that I do not believe Microsoft intend to fix.[3] No solvers disagreed on the outcome of any benchmark: while some failed to produce a result, no solver reported a property violation where another reported successful verification, and vice versa.

[3] Specifically, an additional 112 tests time out and 30 more crash in out-of-memory conditions when ESBMC is run over the SV-COMP benchmark suite with Z3 4.3

Directory	Testname
pthread-ext	40_barrier_vf_false.i
pthread-ext	41_FreeBSD__abd_kbd__sliced_true.i
ldv-linux-3.4-simple	32_7_cilled_false_const_ok_linux-32_1-drivers–net–wireless–p54–p54usb.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.c
pthread-ext	42_FreeBSD__rdma_addr__sliced_true.i
ldv-regression	test_union_cast-1_true.i
ldv-linux-3.4-simple	32_7_cilled_false_const_ok_linux-32_1-drivers–staging–keucr–keucr.ko-ldv_main1_sequence_infinite_withcheck_stateful.cil.out.c
pthread-ext	46_monabsex2_vs_true.i
ldv-consumption	32_7a_cilled_true_linux-3.8-rc1-32_7a-drivers–net–ethernet–sfc–sfc.ko-ldv_main2_sequence_infinite_withcheck_stateful.cil.out.c
ldv-consumption	32_7a_cilled_true_linux-3.8-rc1-32_7a-drivers–net–ethernet–sfc–sfc.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.c
pthread-ext	43_NetBSD__sysmon_power__sliced_true.i
ldv-regression	test_union_cast.c_true_1.i
ldv-linux-3.4-simple	32_7_cilled_false_const_ok_linux-32_1-drivers–usb–storage–usb-storage.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.c
ldv-regression	callfpointer.c_false.i
pthread-ext	44_Solaris__space_map__sliced_true.i
ldv-regression	test_union_cast.c_true.i
ldv-consumption	32_7a_cilled_true_linux-3.8-rc1-32_7a-drivers–net–ethernet–sfc–sfc.ko-ldv_main3_sequence_infinite_withcheck_stateful.cil.out.c
ldv-regression	test_union_cast-2_true.i

TABLE 4.1: SV-COMP'14 benchmarks omitted from SMT solver comparisons due to parsing and conversion errors

Z3	Boolector	MathSAT	CVC4	Yices
4.0	1.5.118	5.2.8	1.3	2.2.0

TABLE 4.2: Solvers used in this evaluation, with version numbers

The number of tests failing due to crashes or timeouts, and the number that successfully report a result are given inTable 4.3. A crashing test is one where ESBMC terminates after receiving the `SIGSEGV` or `SIGABRT` signals, caused by either a programming error in ESBMC, or the process running out of memory. As all the tests are run in exactly the same way, any errors in ESBMC will present themselves in all runs of the benchmark. Within the set of crashing tests, I identify those that crash due to running out of memory by searching the program output for the C++ `std::bad_alloc` exception, as well as other error strings printed by each solver when they fail to allocate memory.

These results appear to show Boolector, MathSAT and Z3 sharing a small set of of tests that always crash or run out of memory. Of the benchmarks on which these three solvers crash, 151 of them are in common. CVC and Yices however, crash on a much greater number of tests.

In terms of speed, only CVC shows a substantial reduction in performance relative to the rest of the solvers, timing out in more than 15% of the 2877 total benchmarks. Balancing across both memory consumption and timeouts, Boolector comes out as the best performing solver, producing a result in the greatest number of tests (2471 out of 2877). Z3 is not far behind however, with only 23 fewer completed benchmarks.

Solver	Crashes (OOM)	Timeouts	Completed
Boolector	164 (44)	225	2471
CVC4	235 (65)	470	2155
MathSAT5	165 (70)	284	2410
Yices2	248 (45)	223	2389
Z3	164 (54)	248	2448

TABLE 4.3: Crashed, timed out and completed tests for the SV-COMP'14 benchmarks, with an unwind bound of 8, and different solvers. The (OOM) field represents the number of crashes that were caused by out-of-memory conditions.

Table 4.4 contains a breakdown of the cumulative amount of time taken to verify the benchmarks in a directory of the SV-COMP'14 repository [73]. The timing resolution for each test is one second. I group these by the containing directory rather than the categories that SV-COMP defines as all the benchmarks in each directory are contributed by a single source, and it will be easier to observe the different characteristics of different contributions this way rather than by aggregating across categories. To make this a fair comparison between solvers, those tests where any solver times out or crashes are omitted (and are covered by Table 4.3). The results themselves are a mixed bag: no solver dominates the others overall, although Boolector delivers the fastest time in 17 of the 26 directories, 5 of which are shared with other solvers. The total for all directories shows that Boolector is the fastest overall, and in two directories (bitvector and eca) is 50% faster than any other solver. Yices achieves this in the loops directory, apparently because of its very swift ability to identify unsatisfiable formulae.

This data shows that Boolector compares favourably with the other SMT solvers that I have tested, and as a result ESBMC will be using Boolector at the next Software Verification competition, possibly as a default solver. Given that some solvers are better than others for certain classes of problem, it may also be effective to identify which solver is best for a particular problem, if reasonable heuristics can be developed. This comparison of SMT solvers is a good starting point for identifying such heuristics, but more work is required.

4.2 Improving accuracy and efficiency of memory references

As mentioned previously in this chapter, a particularly complex portion of ESBMC's encoding of C expressions to SMT relates to pointers: specifically, the fact that one may address any portion of a data object in a pointer (subject to certain rules), and dereference that pointer to access the corresponding piece of data. As an example, one may cast a pointer to an integer into

Directory	Num tests	Z3	Btor	MSAT	CVC	Yices
bitvector	35	115	**30**	107	1620	192
bitvector-regression	13	3	**1**	3	2	**1**
ddv-machzwd	13	**3**	**3**	5	5	4
eca	35	1204	**320**	2068	8797	902
heap-manipulation	1	**6**	7	32	395	17
ldv-commit-tester	24	93	87	327	186	**78**
ldv-consumption	38	**1067**	**1067**	1074	1285	1072
ldv-linux-3.0	11	1110	948	**931**	**931**	937
ldv-linux-3.4-simple	1076	10783	**10732**	10975	12087	10862
ldv-regression	36	3	3	7	8	**2**
list-ext-properties	17	2	**1**	7	11	3
list-properties	12	5	**4**	11	31	**4**
locks	13	**2**	**2**	**2**	46	7
loops	58	12	65	25	201	**1**
memsafety	14	9	**4**	514	177	5
memsafety-ext	8	3	**2**	5	4	3
ntdrivers	1	**1**	**1**	**1**	**1**	2
ntdrivers-simplified	10	**3**	5	6	94	**3**
product-lines	485	154	148	171	9032	**125**
pthread	18	3289	**1819**	2874	3857	2806
pthread-atomic	8	49	**42**	50	51	51
pthread-ext	20	12405	**9902**	12483	12459	12466
recursive	19	51	7	13	581	**5**
seq-mthreaded	100	**104**	111	886	3384	137
ssh-simplified	25	85	**32**	34	4402	85
systemc	12	126	**35**	58	2047	51
TOTAL	2102	30687	**25378**	32669	61694	29821

TABLE 4.4: Total cumulative verification time, per directory in SV-COMP'14, of all benchmarks that successfully completed verification for all solvers. Time measured in seconds. The fastest solver per directory is typeset in boldface. All benchmarks in the "ssh" directory in the benchmark suite timed out with the CVC solver, and thus no results are presented

a character pointer, increment it, and then dereference the pointer to read a byte internal to the integer.

The C specification is very tolerant of such data manipulation, and requires that every data object valuation has a consistent *byte representation*, a sequence of byte values that represents that data object valuation. C programs are allowed to access any part of a data object through a character pointer to examine this byte representation. The C specification does require, however, that all memory accesses must lie within the bounds of the data object, and that pointers must be sufficiently aligned for their type, i.e. a 4 byte integer pointer must always be 4-byte aligned.

CBMC's [53] encoding to SAT was very amenable to this behaviour, as all data objects were ultimately reduced to a vector of boolean values, which could be individually addressed and operated upon without affecting the rest of the data object—essentially, a one-bit byte model. LLBMC [121] represent all memory as a flat byte array, a technique they admit is not particularly

```
struct { int foo; int bar; } xyzzy;
int *p;

if (nondet_bool()) {
        p = &xyzzy.foo;
} else {
        p = &xyzzy.bar;
}

*p = 0;
```

FIGURE 4.1: A piece of code with a nondeterministic pointer offset valuation

efficient. ESBMC's SMT encoding does not have either such memory models however, and uses a typed representation of data objects. Accesses to the internals of a composite data object require packing and unpacking of the type: consider an integer inside an array, inside a tuple. To access an arbitrary byte within it, one must project the field from the tuple, select the element from the array, and then encode an SMT extract expression against the integer.

The most pathological circumstances are those where the offset into the data object is unknown, and potentially a range of values. This can be caused by any program nondeterminism, for example in Figure 4.1, where the pointer `p` may point at one of two locations in the `xyzzy` data object. ESBMC cannot statically determine exactly where `p` may point, and so it considers `p` as potentially pointing at any byte of `xyzzy`, including locations where an access will range across the struct field boundaries. Worse, assignments must consider an assignment to any location of the data object, including aggregate variables such as structures or arrays.

Such circumstances require complicated SMT expressions to be built, which versions of ESBMC prior to v1.22 did not implement completely, leading to crashes and sometimes incorrect results. For SV-COMP'14, I tested two approaches to resolving this problem. Both of the approaches fail to preserve values assigned to padding between elements of structures or arrays, in violation of the C specification that requires that data objects can be treated as arrays of characters if necessary (although this causes their values to become indeterminate). This is not behaviour ESBMC has supported before, however, and so I do not consider it further.

4.2.1 Unaligned byte-accurate memory model

My initial attempt at implementing such memory references tried to produce a precise expression for each possible memory access. For dereferences where the offset into the data object could be statically determined, the addressed value is projected out of the containing data object, or in the case of a write when the data object is updated with the relevant field modified. The most pathological case found in this approach was where a large memory access was performed that overlapped several smaller data objects, for example an unaligned 32 bit read in an array of 16-bit integers, in such a way that data was read from 3 elements of the array.

For memory references where the offset into the data object is indeterminate, I initially attempted to convert the data object to a single bitvector and then update a parameterised portion of it. However this approach swiftly consumed large amounts of memory and frequently prevented ESBMC from completing verification. Instead, for these references I created an array of bytes, and converted the data object into its byte model representation within that array. Reading data is then performed by selecting the bytes corresponding to the desired offset, then coercing them back into the desired datatype. If a write is to occur, the data object must be reconstructed from the byte representation array.

4.2.2 Align-guaranteed memory mode

While considering the previous attempt to implement memory references in SMT, I observed that the memory references that I consider the most complex were those that violated C memory alignment rules. Alignment rules require that any pointer variable must be aligned to at least the alignment of the pointer type—so for example, an integer pointer's value must be aligned to at least 4 bytes, for 32-bit integers. This causes pathological references, such as the 32-bit read over an array of 16-bit values described above, to be undefined by the C specification, and thus to be program errors.

To take advantage of this, I first encode property assertions when dereferences occur during symbolic execution, to guard against executions where an unaligned pointer is dereferenced. This is not as strong as the C standard requirement, that a pointer variable may never hold an unaligned value, but it provides a guarantee that any pointer dereference will either be correctly aligned or result in a verification failure. This means the construction of memory references in SMT need only consider offsets into the data object that are aligned.

Further, I arrange the memory layout within C structures so that each field of the structure has the greatest possible alignment guarantee.[4] This ensures that the largest possible type in ESBMC can only ever reference one structure field without breaking alignment rules. Smaller types also cannot create a reference to more than one field in a data object without violating alignment rules. This constructively prevents any memory reference from legally referring to more than one structure field at a time. Arrays, however, may still have more than one element accessed during a dereference (consider a character array aligned on a 64-bit boundary accessed by a 64-bit integer pointer). This arrangement is permitted by the C specification as the padding between structure fields is an implementation defined matter.

At the SMT level, as with the previous encoding any statically determinable offsets into data objects are directly constructed. Indeterminate offsets are constructed considering the alignment guarantee of the reference: because memory references cannot cross fields we can consider each structure field individually, and the dereference either evaluates to the field itself or bytes within

[4]The largest variable that ESBMC supports is a 64 bit integer or double precision floating point number, so the greatest alignment in ESBMC is 8 bytes.

it: no fields are composed together, as they were with the previous approach. Arrays with a smaller type than that of the memory reference may have multiple elements accessed, but alignment rules guarantee that the access begins at the start of an element, not the middle, and includes a whole number of elements.

4.2.3 Comparison and evaluation

The primary purpose of these memory models is to reduce the number of SV-COMP benchmarks containing memory references that ESBMC could not encode. Each of them eliminate the 80 benchmarks in SV-COMP'14 that were signalling that an unimplemented memory reference format was encountered. Of more interest however is which of these two memory models is the more efficient. Table 4.5 shows the aggregate results when each model is applied to the SV-COMP'14 benchmarks, using the command line given in Section 4.1. Note that the versions of ESBMC used in this section are older than those used through the rest of this thesis, as the development branches of the two memory models were not been updated after they were completed. Accordingly, the results here are not directly comparable with any other results.

ESBMC configuration	Timeouts	Crashes (OOM)	Completed tests
Byte-precise model	309	150 (71)	2387
Aligned-access model	259	113 (49)	2488

TABLE 4.5: Verification results when running ESBMC with the byte precise and aligned memory models. "Timeouts" counts the number of tests that terminated after running out of the allocate 15 minutes, "Crashes" the number of tests that crashed and (OOM) the number that crashed in out-of-memory conditions, "Completed tests" the number of tests that completed verification. These totals omit 17 tests that ESBMC could not parse, see Section 4.1.2

These results show that there is a notable improvement in both the number of tests that time out and those that run out of memory, between these two memory models. In particular, 100 more tests completing verification is most welcome.

The addition of the alignment properties used by the align-guaranteed memory model can be helpful for detecting undefined behaviour, which is ESBMCs primary task. These properties are not, however, of any use during the Software Verification competition, as the competition assumes that all of the benchmarks being verified already have correctly formed memory accesses.

4.3 Guard abstractions

Some of the largest SMT expressions produced by ESBMC are those evaluating the guard of execution paths—i.e., whether under the current valuation of the formula, the path is followed or not. The cause is the branching and merging of different paths through the program. Consider the code in Listing 4.2, and the merging of different paths of execution through the program.

```
int x = nondet_int();
int y = 0;
int i;

for (i = 0; i < x; i++) {
        y++;
}
```

FIGURE 4.2: A loop with exit branches that depend on nondeterminism

When paths are merged, the guard of the resulting path is the disjunction of those paths merged in. This technique is followed by CBMC, ESBMC and LLBMC. LLBMC presents [121] it in the following manner, in terms of basic blocks rather than instructions. The guard $G(b)$ of block b is computed recursively using the predecessors $P(b)$ of the block b, and the condition $t(b', b)$ on entering block b from block b'. Then,

$$G(b) = \bigvee_{b' \in P(b)} (G(b') \wedge t(b', b)) \tag{4.1}$$

ESBMC handles this at a per-instruction level, but using the same formula. Effectively, the guard is a combination of all the guards that must be true for a particular path to be explored. ESBMC is only able to statically determine the truth of guards in a small number of cases, and not at all in the presence of nondeterminism, and as a result the guard expression can become large. In the worst case scenario, the guard can grow exponentially, when loops with conditional exits that cannot be statically determined are composed. The guards for all paths through the first loop will be encoded in the guard for each path in the second loop.

As the guard is used in every property assertion and assumption in the program, as well as *phi* functions when merging paths, reducing the size of the guard accumulated could reduce the size of the formula significantly. One way to counteract this would be to, once the guard reaches a certain size (measured by number of sub-expressions), assign the guard expression to a new boolean SMT variable, and use that new variable as the path guard from then on. This trades the size of the guard expression for additional variables and thus state space in the program. The authors of LLBMC [121] suggest that this is best implemented by creating a new boolean variable on entry to every basic block, representing the guard for that block. This is the approach that I have used when evaluating the optimisation in ESBMC.

4.3.1 Evaluation

To evaluate the guard-abstraction optimisation, I implemented a version where the current guard is assigned to a new symbol upon every control flow merge. Paths through very deep conditional statements may still posses large guards, they cannot however accumulate in size for any long period of time, as control flow merges tend to be frequent. I then tested the feature on the

SV-COMP'14 suite using the command line given in Section 4.1, and compare it against the version of ESBMC used in Section 4.1 (without the guard-abstraction). As this optimisation was proposed by LLBMC [121] and they use the Boolector solver, I also use that solver for the evaluation. The results are presented in Table 4.6.

ESBMC configuration	Timeouts	Crashes (OOM)	Completed tests
No guard-abs	225	164 (44)	2471
Guard-abs	223	148 (28)	2489

TABLE 4.6: Verification results when running ESBMC with and without the guard optimisation. "Timeouts" counts the number of tests that terminated after running out of the allocate 15 minutes, "Crashes" the number of tests that crashed with (OOM) the number that crashed in out-of-memory conditions, "Completed tests" the number of tests that completed verification. These totals omit 17 tests that ESBMC could not parse, see Section 4.1.2

Here we see that with the guard optimisation enabled, the number of timeouts decreases by a negligible amount, not worthy of note. The number of test cases that result in out-of-memory conditions however decreases by 16, a useful outcome from this optimisation.

4.3.2 Addition of slicing optimisation

The size of the guard expression has a secondary effect on ESBMC's performance outside of the SMT solver. ESBMC can use the "slicing" optimisation inherited from CBMC to reduce the number of assignments in the program. Slicing [160] is a mechanism for eliminating redundant or irrelevant portions of a program. In ESBMC, this is applied to the SSA program (Section 2.4.2) before it is encoded to SMT to reduce the number of variable assignments, by identifying variables not used in the evaluation of any property assertion. Every expression in the SSA program must be examined to achieve this, and when program guards are large, the amount of time consumed by the slice optimisation becomes very large. For example, enabling the slice optimisation when running the SV-COMP'14 suite as described in Section 4.1 increases the amount of time consumed by 40%.

This raises the prospect that the guard abstraction optimisation could compliment the slicing optimisation, through its reduction of expression sizes. To evaluate this, I re-ran the tests above with the slicing optimisation enabled, the results of which are presented in Table 4.7.

ESBMC configuration	Timeouts	Crashes (OOM)	Completed tests
No guard-abs, Slicing	346	98 (42)	2416
Guard-abs, Slicing	218	85 (20)	2557

TABLE 4.7: Verification results when running ESBMC with and without the guard optimisation, with the addition of the slicing optimisation. "Timeouts" counts the number of tests that terminated after running out of the allocate 15 minutes, "Crashes" the number of tests that crashed with (OOM) the number that crashed in out-of-memory conditions, "Completed tests" the number of tests that completed verification. These totals omit 17 tests that ESBMC could not parse, see Section 4.1.2

With both optimisations enabled, ESBMC successfully completes a greater number of tests, with 2557 of the benchmarks successfully reporting a verification result. This is 86 more results than the original configuration of ESBMC, without guard abstraction or slicing. The total number of timeouts, crashes, and out-of-memory conditions is lower than with any of the optimisations disabled.

Examining the results closer, we see that in normal ESBMC the slicing optimisation substantially increases the number of tests that time out, while reducing the number of crashes. This is in line with expectations: large program guards take a long period of time to slice, and that without slicing the size of the SMT formula is larger, leading to greater memory consumption and crashes. When the guard abstraction is enabled a significant reduction in timeouts is observed, caused by the sliced expressions being much smaller.

I cannot claim any credit for the reduction in crashes caused by the slicing optimisation, but I have demonstrated here that the guard abstraction optimisation reduces the size of the SMT formula leading to a reduction in memory consumption, and compliments slicing by reducing expression sizes leading to faster verification and fewer crashes when the two are composed.

4.4 Conclusions

In this chapter I have examined the performance of different SMT solvers when applied to the SMT formulae produced by ESBMC, and evaluated two optimisations of the SMT formula encoding.

The SMT solvers tested had a mixed set of results, in which only CVC and MathSAT proved to be consistently slow solvers. Boolector, Z3 and Yices all demonstrated subclasses of problems where they verified problems faster than the other solvers, with Boolector having the best performance overall, running the fastest in 11 directories out of the 27 in the benchmark suite, and running in the smallest amount of time as an aggregate over all of the benchmark suite. This is in line with Boolectors success in the QF_ABV category of SMT-COMP'14 [65], and a compelling reason to use Boolector as the default solver in ESBMC.

I have also identified a class of problem that ESBMC handles poorly (the construction of memory references), and evaluated two different approaches to solving this problem. When applied to the SV-COMP'14 benchmarks, it is clear that the alignment-guaranteed memory reference model, with smaller and less complex SMT expressions, leads to reduced verification time and memory consumption.

Finally, reducing the number of SMT expressions in the formula through abstraction of the path guard is shown to reduce the amount of verification time required, when used in conjunction with another previously unfeasible optimisation.

In total, the work presented in this chapter results in an improvement from 2448 of the SV-COMP'14 benchmarks being verifiable (when using Z3, see Table 4.3), to 2557 being verifiable (Table 4.7, which includes the align-guaranteed memory model).

An important conclusion to draw from this work, aside from the techniques that I have shown to be effective optimisations, is that there is still significant scope for the verification of software with SMT solvers to be improved. The performance of model checking tools it not simply related to the way in which the piece of software being verified is unrolled, but also the manner in which the underlying formula is encoded.

Chapter 5

Improving the performance of ESBMC for multi-threaded programs

The state explosion problem remains the most significant barrier to verification of concurrent software. In general, the verification task's state space will grow exponentially with the size of the program being verified, and lacking a dramatic theoretical breakthrough it is unlikely that this problem can be reformulated to have a smaller state space. In light of this, I believe the most practical way of increasing the feasibility of verifying concurrent programs is the development and application of optimisations, to reduce the time required to explore the interleavings produced by composing concurrent threads (see Section 2.2.1). These can be optimisations that increase the speed at which individual states can be checked, or reduce the number of interleavings in a program that need to be checked.

In this section I examine four optimisations I have implemented to improve the performance of model checking concurrent programs with ESBMC, and explain their advantages and empirical results to date. Of these measures, only the first (state hashing, Section 5.2) has been pursued as far as publication [125], and is co-authored with Lucas Cordeiro, and my supervisors Denis Nicole and Bernd Fischer; the work presented here is all my own, however. I evaluate Monotonic Partial Order Reduction [106], which to my knowledge has not been applied to software verification before. I implement incremental SMT solving to increase the speed at which states can be verified, and use the availability of the SMT formula during symbolic execution to guide exploration of the state space, a mechanism which has not been previously applied to concurrent software in the literature.

More optimal encodings of the SMT formula that ESBMC creates during verification, and optimisations internal to ESBMC itself would improve the performance of verifying concurrent software: they apply however just as well to single threaded verification. Such optimisations are presented in Chapter 4, while this chapter covers optimisations only relevant to concurrent software.

Before studying optimisations, it is first important to understand how ESBMC explores concurrent programs. When symbolically executing any piece of code in a multithreaded context, ESBMC checks whether the most recently executed instruction accesses any variable that is shared with other threads. If it does, then ESBMC is at a *context switch point*, where it may decide to switch execution to some other thread. The reachability tree (Section 2.2.1) is explored by examining the path taken by each thread from a context switch point, and any subsequent context switch points are found. Some of the literature on concurrent software describes this in terms of *transitions*, where a program has some set of state transitions that modify the program state, and state exploration occurs by composing transitions. In ESBMC, such a transition would correspond to the path of a thread execution from one context switch point to another. At each context switch point one transition would be available per thread, corresponding to the path that each thread would take until they encountered the next context switch point.

The concept of an "Instruction" in ESBMC must also be explained: the C specification defines the execution of C in terms of *sequence points* (for example the ";" statement delimiter). All side-effects and operations of a statement are completed before a sequence point is passed. No order of evaluation within a statement is defined, however, giving significant leeway for compilers to re-order when different expressions are evaluated. ESBMC compiles such statements down to individual variable assignment instructions, with the pre-modifications, primary assignment / operation of the statement, and then post-modifications happening in that order. As an assignment is a single instruction that ESBMC atomically executes, some of the concurrent behaviours permitted by C are not visible in ESBMC, for example a variable increment such as `x = x + 1;` will never have an interleaving between the evaluation of `x + 1` and the assignment to x. This does not model the real-world problem of another thread assigning `x` between the expression evaluation and the assignment. However, the command line option `--data-races-check` preprocesses the programs so that each instruction has at most one access to shared program state.

5.1 Evaluating optimisations

To determine how effective an optimisation is, we must evaluate it against benchmarks that provide adequate coverage of the different inputs that the optimisation should handle, and determine what kind of performance improvement it yields. In the context of this chapter, there are two metrics that are relevant:

1. The quantity of time consumed to successfully verify a program.
2. The number of interleavings that are explored to achieve program verification.

While it may seem that time consumed is always proportional to the number of interleavings explored, this is not always true. Some of the optimisations in this chapter result in an overhead on state exploration that is proportional to other factors in the program under test. For example, the overhead of state hashing scales with the number of variables in the program. Such factors may guide the selection of optimisations to apply during program verification.

5.1.1 Benchmark selection

Constructing a well rounded and representative set of benchmarks to evaluate optimisations is difficult. The most serious challenge is that concurrent programs are difficult to characterise. Concurrent C programs have very few operations unique to concurrency—as this thesis focuses on concurrent C software, thread communication via channels is not considered. Aside from the creation and termination of threads, all threads behave as individual sequential programs that have some variables shared between them, making concurrent behaviours dependant on program variables only.

Considering the concurrent operations above, I will measure a program by the number of threads that exist within it, which is potentially unbounded if the program can feature an unbounded loop that creates threads. Additionally I will measure the number of variables that are shared between threads in the program. The set of benchmarks picked should have a reasonable distribution of threads and variables.

I would wish to measure the number of instructions / expressions that access such shared variables, but measuring this amounts to a model checking problem in itself.

The actual algorithms used in the concurrent program are also difficult to characterise. Concurrent behaviours are controlled by the value of the shared variables in the program. There is however, the matter of synchronisation, which comes in the form of *mutual exclusion* and *condition variables*. As covered broadly in Section 2.2.1, concurrent programs sometimes require exclusive access to a resource, and achieve this by synchronising with other threads on a lock, which only one thread can hold at a time. Broadening this concept, we can think of synchronisation as being blocks of code in a thread that require certain other threads to be in a particular state, or in particular blocks of code themselves. The benchmarks we use should have a variety of such synchronisation behaviours present.

ESBMC itself is restricted to working with concurrent programs that use the `pthread` [102] standard API for the creation of threads and synchronisation between them. A set of benchmarks that features reasonable coverage of the `pthread` API should exhibit the concurrent operations mentioned above. ESBMC supports all the most commonly used features of `pthread`, the exceptions being read/write locks, thread attributes,[1] and the `pthread_cancel` facility.

Turning our attention to finding or designing benchmarks themselves, there has been a lack of pre-existing test suites in the literature that are suitable for software verification, and many

[1]Many of which control the operating system scheduler and are thus irrelevant to model checking

benchmarks published are "toy" programs used to validate research prototypes [31]. The International Competition on Software Verification [31, 32, 33] (SV-COMP) has in recent years attempted to rectify this by collecting benchmarks used by software verification tools, and normalising them so that different tools can be compared across a common set of benchmarks. Of the 2868 [33] benchmarks, some 81 are part of the Concurrency category, with benchmarks in the `pthread`, `pthread-ext`, and `pthread-atomic` directories in the SV-COMP repository [6].

The concurrency benchmarks in SV-COMP are collected from three sources: from ESBMC [68] in the `pthread` directory, from the CProver team [87] in `pthread-ext`, and Threader [137] in `pthread-atomic`. I consider those benchmarks that feature in the 2014 competition. The benchmarks themselves are described qualitatively below:

- `pthread` (21 benchmarks): Roughly half the benchmarks have only two threads, but the rest range from three to fifteen threads. There is an average of three shared variables between threads, but half of the benchmarks have a large arrays of integer values that threads modify. Almost all files make use of `pthread_mutex` and `pthread_join` for synchronisation, however there is only one use of a `pthread_condvar`.
- `pthread-ext` (45 benchmarks): Almost all of these benchmarks have an infinite loop generating threads, making the number of threads explored dependant on the unwinding bound chosen. No `pthread` synchronisation functions are used.[2] Instead, synchronisation primitives are recreated and tested using three or four shared variables and certain guarantees about atomicity. Various mutual exclusion algorithms are implemented (i.e. Dekker, Peterson) and a multithreaded heap.
- `pthread-atomic` (10 benchmarks): Most of these benchmarks take two threads and implement a mutual exclusion algorithm, some using `pthread_mutexes`. Between three and seven shared variables are typically used.

The coverage of the `pthread` API, number of threads present in the benchmarks, and amount of shared state (ranging from one or two variables to large arrays of shared values) in the SV-COMP concurrency benchmarks are sufficient to evaluate optimisations for concurrent program verification, according to the criteria discussed above. There are two shortcomings: firstly that none of the benchmarks are of a very large size (all save one are less than two hundred lines long). This, unfortunately, is a matter of necessity, as the state explosion problem ensures that very large programs cannot be feasibly verified, and thus they do not appear in the verification competition. The second shortcoming is that the benchmarks are not necessarily balanced between different attributes, and so aggregations of performance measurements may mask subclasses of problems that are negatively affected. This can be countered by examining the performance outliers as well as performance aggregates.

[2] The only `pthread` function call made is to create threads, via `pthread_create`

On the whole, I believe the SV-COMP benchmark suite to be reasonably well rounded. The primary reason for selection however is that there are no other suites of concurrent benchmarks that are suitable for verification, large enough, and have a breadth of behaviours. Those that could be considered already make up part of the SV-COMP benchmark suite.

Six of the benchmarks cannot be parsed by ESBMC; this is due to the `_Thread_Local` storage qualifier, recently introduced in the C11 standard [7], which ESBMC does not yet support. Rather than modifying the benchmarks, they were instead omitted from the benchmarks used in this chapter. The list of omitted benchmarks is in Table 5.1. All were in the `pthread-ext` directory.

40_barrier_vf_false.i	41_FreeBSD__abd_kbd__sliced_true.i
42_FreeBSD__rdma_addr__sliced_true.i	43_NetBSD__sysmon_power__sliced_true.i
44_Solaris__space_map__sliced_true.i	46_monabsex2_vs_true.i

TABLE 5.1: Tests omitted from SV-COMP concurrency benchmark suite due to ESBMCs inability to parse them

Additionally, seven benchmarks yielded verification times of less than two seconds, no matter what unwind bound and context bound were given. They are listed in Table 5.2. These benchmarks only contained two threads and no program nondeterminism—only the order in which threads are interleaved can vary. As a result, ESBMC's constant propagation allows it to find explicit valuations for all variables in the program at all times, and as a result unfeasible interleavings are statically identified and pruned. Several do not have any loops that contain accesses to shared variables, meaning that the state space of all interleavings does not grow with the unwind bound (and is thus finite). Of the two that do (19_time_var_mutex_true.i and 23_lu-fig2.fixed_true.i), the number of interleavings grows roughly linearly with the unwind bound. I decided to omit all these tests from my experiments too, because while they may benefit from performance improvements as a result of optimisations, with verification times in the order of seconds it would be difficult to be confident that any speedup was not the result of timing variations in the test setup.

18_read_write_lock_true.i	19_time_var_mutex_true.i
23_lu-fig2.fixed_true.i	bigshot_p_false.i
bigshot_p_true.i	bigshot_s_false.i
bigshot_s_true.i	

TABLE 5.2: Tests omitted from SV-COMP concurrency benchmark suite due to extremely small verification times and limited state space

5.1.2 Test setup

We now consider how the optimisations featured in this chapter will be evaluated. The first and most significant decision about the execution of these benchmarks is that they will be *context bounded*, i.e., there will be a limit enforced on the number of times thread context switches will be permitted (see Section 2.2.3). This is necessary because, in initial training runs, even at very low unwind bounds the majority of the benchmarks had too many thread interleavings to explore, and used unfeasibly large amounts of time (i.e. hours) to enumerate them all.

Applying a context bound immediately makes the verification incomplete, as we are using a bounded model checker, however the program unwinding bound already makes the verification incomplete. The optimisations to be evaluated should not be significantly affected by this, as context switches will be being explored at all the same points in the program, but less frequently. Only variable valuations that are only found through a large number of context switches will be unreachable.

The Software Verification competition defines the environment and conditions under which its benchmarks are to operate, which are honoured in this evaluation too. For the concurrency category, the **only** property to test in the program is the reachability of the program label `ERROR`. SV-COMP also enforces a fifteen gigabyte memory limit, and fifteen minute CPU-time limit. Here, I enforce the memory limit (due to operational constraints), but extend the timeout to be four hours to give the best chance of the verification completing. I am aiming to measure the performance impact of optimisations, rather than whether ESBMC can complete verification within the SV-COMP limitations. More details on the actual test environment are below.

Another deviation from the SV-COMP configuration is that in this evaluation, ESBMC does not exit when a counterexample is found. Instead, by using the `--all-runs` option, exploration continues after a counterexample is found, until all interleavings have been explored. This is because we are attempting to measure the optimisation's effect on performance and the number of interleavings found, rather than finding the quickest path to a counterexample.

The command line used to launch ESBMC on the program under test is:

```
esbmc --unwind N --memlimit 15g --timeout 4h
--no-unwinding-assertions --no-assertions
--no-bounds-check --no-pointer-check
--no-div-by-zero-check --error-label ERROR
--tuple-sym-flattener --all-runs --context-switch C
-DLDV_ERROR=ERROR -Dassert=nope -D_Bool=int
--verbosity 0 --quiet
```

It is followed by the path to the benchmark file, and any options required to enable the optimisation being tested. The unwind and context bounds selected are explained below. Breaking down each line,

1. Set the loop unroll bound to N,[3] memory limit, and timeout
2. Disable in-program assertions, and assertions identifying incomplete loop bounds
3. Disable array-bound and pointer violation checks
4. Disable divide-by-zero checks, and make reaching the label ERROR a property violation
5. Configure an internal option, add `--all-runs` and configure the context bound to C[4]
6. Preprocessor workarounds to avoid problems with certain SV-COMP benchmarks
7. Reduce the amount of output text produced by ESBMC during its verification run

ESBMC also possesses two mechanisms for exploring multithreaded state space. Interleavings between threads are always explored explicitly, but the SSA program for each interleaving can be encoded either to the same SMT formula, or one formula per interleaving (see Section 2.2.2). Previous research [68] has shown the latter to be the most efficient, and so I use that technique (the depth first search encoding) throughout this chapter.

The actual test runs are executed on the Iridis 4 supercomputer hosted at the University of Southampton, a compute cluster of 750 nodes, with 2x 8-core Intel "Sandybridge" processors and 64GB of memory per node. Each test was given a time limit of 4 hours, and was executed on a node with three other instances of ESBMC running these tests (the maximum achievable while allocating 15GB of memory per process). The version of ESBMC used was b62e4ec7,[5] and was compiled with GCC 4.8.2 on a Ubuntu 14.04 machine. Iridis 4 itself runs RedHat Enterprise Linux 6.3 on compute nodes.

The solver used for all of these tests is Z3 [72]. This is ESBMC's default solver, and the one for which ESBMC's support is most mature and robust. Z3 supports all of the features required in this chapter (i.e., incremental solving, see Section 5.4.3.1) which many solvers do not. In addition, Z3 consistently won many categories at the SMT-solver performance competition, for several years in a row [63]. While Chapter 4 showed that Boolector is generally faster than Z3, it does not have full support for all SMTLIB2 features, in particular the SMT push and pop facilities used later in this chapter.

In the analysis I call one execution of ESBMC against a single testfile with a particular bound configuration a "run", while "benchmarks" refers to the set of test files that make up the SV-COMP concurrency benchmarks, including all the test runs in all bound configurations.

[3] i.e., the number of times to unroll each loop
[4] i.e., the maximum number of context switches to allow
[5] A git hash, from which the exact version of ESBMC used can be retrieved

5.1.3 Interpreting results

In practice, it is difficult to present the results of this chapter in the form of a graph or illustration. As the benchmarks are diverse, there is no particular unwind bound or context bound where all the benchmarks successfully complete verification, save at excessively low unwinding bounds. Additionally, at higher bounds some tests run out of memory and terminate early, making them incomparable with other tests run with the same bound configuration. I have tested all the benchmarks with a range of unwind and context switch bounds, from one to five loop unwinds and one to seven context switches permitted, but here I present three subsets of that data.

Firstly, for each benchmark I take the unwind and context bound configuration that, when using ESBMC without optimisation, leads to the *longest running* verification that successfully completes. These are shown in Table B.1. Being the longest run for each benchmark, any improvement (or penalty) of the optimisations examined in this chapter should have the most impact, of the available runs. These are presented as tables ordered by performance improvement and then benchmark name, to ease comparison.

Secondly I examine the set of runs that complete verification when optimised but either crash or time out in the unoptimised version of ESBMC, and vice versa. These are referred to as newly-succeeding and newly-failing (as a result of the optimisation) respectively. Finally, the set of all run configurations where both the optimised and unoptimised version of ESBMC complete verification (the "commonly successful" set) is examined for any statistical oddities over all the successful runs. This should identify any outliers in the performance data that may be significant.

Any truly successful optimisation should cause a large number of test runs that previously timed out or exhausted memory to successfully verify, and not cause any test runs to exhaust their resources where they would not have without the optimisation. Moving the boundary of what can and cannot be verified is one of the ultimate aims of these optimisations. It is also possible for optimisations to increase ESBMCs performance when solving certain problems but not others. Such optimisations can still be useful for a subset of verification problems.

The tables of results for this chapter can be found in Appendix B.

5.2 Symbolic state hashing

State hashing has already been described in Section 2.2.3.2, in the context of explicit state model checkers. To recap, the approach is to compute a checksum or summary of a state that has been explored, and then index the set of explored state checksums. Given any state, we can summarise it and then efficiently look up whether it has been explored before along a different path, and if so the current path need not be explored. Within model checkers such as SPIN, the use of a summary or hash eliminates the need to perform expensive state comparisons or compression, and makes the check for explored-states a task with O(1) complexity.

This optimisation can also be useful in symbolic bounded model checking of concurrent programs. If, in ESBMC, we reach a state where a context switch may be taken (referred to here as a "node"), and all shared and thread-local variables and program counters are the same as another explored node, then only one such identical node needs to be explored, as the reachability subtrees of identical nodes will be the same. As an example, consider a simple multithreaded C program shown in Figure 5.1 and its corresponding reachability tree shown in Figure 5.2. The reachability tree consists of the nodes ν_0 to ν_{16}, where each node is defined as a tuple $\nu = (A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n)_i$ for a given time step i. Here, A_i represents the currently active thread, C_i the context switch number, and s_i the current (global and local) state. Further, for each of the n threads, l_i^j represents the current program counter of thread j and G_i^j represents the control flow guards accumulated in thread j along the path from l_0^j to l_i^j (although these are not shown in Figure 5.2). Notice that the transitions originating from node ν_1 as those originating from ν_7, produce the same program states. When we explore the node ν_7, we can simply eliminate the transitions that originate from it—provided that we realise that we have already explored another identical node.

Storing the symbolic state of the program is an impractical task. The valuation of variables in the program can be ranges of values, constrained by the entire set of past assignments and paths taken in the program, thus requiring the whole history of an execution to be stored. Comparing two symbolic states is expensive for the same reasons. Thus, maintaining a set of hashes of the explored symbolic states at nodes would be a much more efficient way of identifying and removing duplicate states.

```c
#include <pthread.h>
int x=0, y=0;
void t1(void* arg) { x++; }
void t2(void* arg) { x++; }
void t3(void* arg) { y++; }
int main(void) {
  pthread_t id1, id2, id3;
  pthread_create(&id1, NULL, t1, NULL);
  pthread_create(&id2, NULL, t2, NULL);
  pthread_create(&id3, NULL, t3, NULL);
  return 0;
}
```

FIGURE 5.1: A simple multi-threaded C program.

5.2.1 Hashing symbolic states

State hashing is not simple to achieve with symbolic model checking, again due to the fact that variable valuations are defined by the history of assignments and constraints in the SSA program (Section 2.4). To counter this, we compute a hash for each program variable's value,

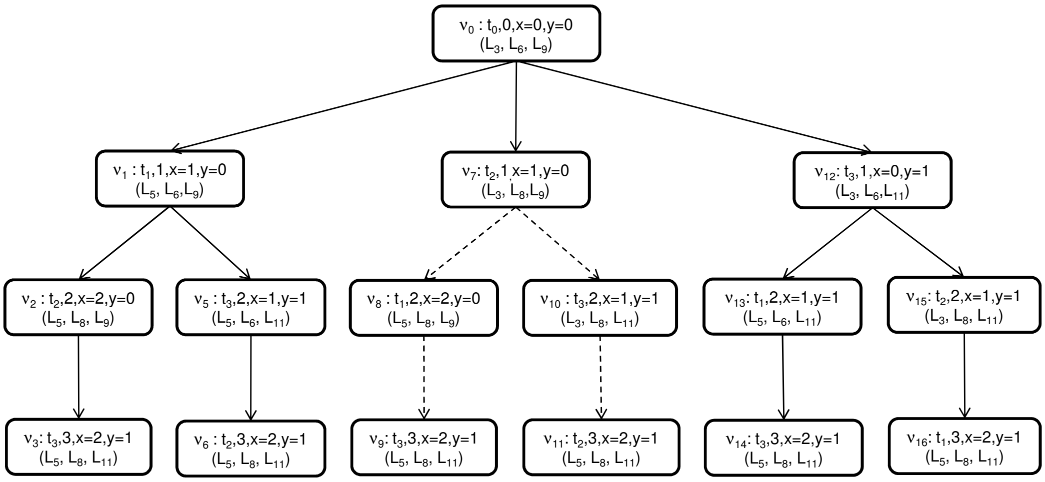

FIGURE 5.2: Reachability tree for the program in Figure 5.1. Dashed edges represent transitions that can be eliminated by the state hashing technique.

and include the hash value in new assignments, effectively making each hash depend on variable history. These variable hashes are then combined to make a node-level hash that represents a particular RT node, resulting in a two-level hashing scheme.

The node-level hash is created by taking the variable-level hashes of all variables in the current node and concatenating them, together with the program counter values of all existing threads, in a consistent order into a single data vector. This vector is then fed to a hashing function. Variable-level hashes are more complex. For each assignment encountered in the RT exploration we calculate a hash of the right hand side expression and record it against the left hand side variable name. This hash is created by serialising each operator and value in the expression to a data representation (i.e., a series of bytes) into a vector, which is then hashed.

For example, Figure 5.1 contains several assignments to the global variable `x` using the ++ operator (converted to an addition and assignment internally). ESBMC automatically performs constant propagation and effectively converts the example to an explicit state check (i.e., none of the variables in this program have an indeterminate value at any time). We represent the first serialised increment expression as the text: "(+,(constant(0)),(constant(1)))" This demonstrates one of the simplest encodings of data possible with this method. Any set of operations on constant values can also be expressed in this manner. Such expressions are, however, not yet symbolic—to support this we represent nondeterministic values with a prefix and unique identifier. We also represent the use of indeterminately valued variables in expressions with its current variable hash. To demonstrate this, reconsider Figure 5.1 and assume `x` is initialised to a nondeterministic value. The serialisations representing the two increments of the `x` variable then become: "(+,(nondet(1)),(constant(1)))" and "(+,(hash(#1)),(constant(1)))" where #1 represents the hash value of the `x` variable, i.e. the hash of the first serialised expression. This causes the hash value of the `x` variable to depend on the history (i.e. hash values) of the variables used to calculate its value. Significantly, no thread specific data is encoded in this representation, meaning that the same serialised representation is produced for whichever order of threads increments x. Thus the hash of any variable is a direct product of all nondeterministic inputs, constant values and operators that represent the constraints on the assignment, without the need to store the entire history of assignments.

This method is limited, however, by the ordering of assignments—if the original example in Figure 5.1 had instead a thread that increased the x variable by 2, and another that increased x by 3, then at the end of execution the variable hash of x would be different depending on the thread ordering, even though x ends up with the same value no matter what order of interleavings occur. This also affects arrays (including the heap, which is modelled as an array) and unions. So, only exact symmetry between thread interleavings is identified and eliminated by this approach.

5.2.2 Selection of hash function

As hashing is a lossy abstraction of a node, we risk computing identical hashes for two distinct nodes. Should this occur, one node will be successfully explored and its hash stored; and when the other is explored we will discover its hash in the visited states set, and incorrectly assume it has already been visited. This would cause an unexplored portion of the state space to be discarded.

The metrics that characterise a hash function are its output bitwidth, its performance when hashing data, and the likelihood of producing a collision. The first two factors tend to trade against the last: wider and slower hashes are less likely to result in a collision. However they also consume more resources during verification (memory and time).

For our previous work [125] we chose SHA256 [8] hashes due to its certification for use in cryptographic applications, aspects that assure us the likelihood of collisions is extremely low. It does consume more resources than other hash functions we could have picked, but we decided to act as cautiously as possible.

5.2.3 Experimental Evaluation

Our initial work with state hashing was in the context of Section 3.1, where an LTL monitor thread was being interleaved with a program under test, and a performance optimisation became necessary. The evaluation (Section 3.1.3) showed that performance was improved in all test runs, with a median reduction in verification time of 56%, the maximum 80% and minimum 13%. The exact results and timings are in Table 3.1 on page 59. This would suggest that state hashing is a useful optimisation when verifying concurrent software. However, we must also evaluate state hashing on our chosen suite of benchmarks, using the procedure described in Section 5.1, running ESBMC on each benchmark with a range of context bounds, with and without state hashing.

The performance impact on the longest running test runs is shown in Tables B.2 and B.3. It is immediately apparent that significant improvements in performance have been made, with 32 of the runs having their runtime reduced to less than 10% of the runtime of the corresponding unoptimised test.[6] The mean performance change was a reduction in time consumed by two thirds, the median reduction being a 95% reduction in time consumed. Not all the runs showed improvement however, with three runs having marginally increased runtimes.

Examining the newly-succeeding and newly-failing tests, there are no runs where the unoptimised version of ESBMC completes verification but the state hashing version does not. In comparison, 153 state hashing runs successfully complete where the unoptimised version either runs out of time or crashes. These 153 runs are spread across 42 different test files, coming roughly equally from all directories in the SV-COMP suite.

The set of commonly successful test runs total 1973. Of these, 848 runs show no time difference caused by the optimisation (typically those runs where the bounds are so low that exploration takes only a few seconds). 890 runs complete in a smaller amount of time with state hashing, while 235 take longer. Of those runs where time increases, the mean increase is only 14 seconds, median 1 second, with only a single run increasing runtime by more than 100 seconds. Of those where state hashing reduced time consumption, the mean speedup was 271 seconds, median nine, with 57 runs improving by more than a thousand seconds, and 222 by more than 100 seconds. Additionally, only fifteen benchmarks had runs where the performance reduced by more than ten seconds, while 64 of the benchmarks had at least one run that increased performance by ten seconds.[7]

This is strong evidence that state hashing is an effective optimisation for exploring the state space of concurrent programs, as the majority of tests receive a significant performance improvement, and the effect on those that do not is small. Additionally, the increased number of runs that complete verification with state hashing is an immediate and significant improvement. The improvements are not across the whole benchmark suite, and while the actual performance gains depend on the symmetry encountered in the benchmark under test, we have seen that a majority of the benchmarks benefit from the optimisation.

5.3 Monotonic partial order reduction

As described in Section 2.2.3.1, partial order reduction is the process of classifying transitions within a multithreaded program that are independent of or dependant on transitions in other threads, in order to determine whether pairs of interleavings always compute the same state. Identifying such relations allows us to discard portions of the reachability tree as duplicated. The difficult task in this approach is the classification of transitions.

[6] An elapsed time of zero seconds indicates that ESBMC completed in less than one second; times have not been rounded up

[7] There is some overlap in these figures, which is due to low context bounds not allowing sufficient state space exploration to let state hashing exhibit its effects

Clearly, using a partial order reduction algorithm to reduce the number of interleavings to explore would be beneficial to ESBMC. A recent development in POR algorithms is the Monotonic Partial Order Reduction [106] (MPOR), which Kahlon et al. prove is both sound and optimal, in that it eliminates all redundant interleavings without missing any behaviours that can be exhibited by the program. They then demonstrate a worked example based on the classic dining philosophers problem, manually encoding it to SMT with and without the MPOR algorithm, and evaluate its performance. While eliminating all such interleavings would constitute a substantial improvement in any multithreaded model checking algorithm, it remains to be seen whether the overhead of calculating the POR algorithm outweighs the savings from reducing the number of interleavings explored.

5.3.1 The MPOR algorithm

Briefly, we cover the algorithm that calculates this partial order reduction. We follow the syntax from Section 2.2.3.1 and additionally the syntax described by Kalhon et al. [106], by letting $t_1 <_x t_2$ (where $t_1, t_2 \in T$) denote a path through program x where transition t_1 is taken before t_2. The function $tid(t_i)$ evaluates to the *thread identifier* of the thread executing the transition t_i.

We assume that some underlying dependence / independence relation between transitions exists. The exact meaning is beyond the scope of the algorithm itself, but an example would be two transitions where one reads and the other writes the same piece of data—the order in which they are executed can affect the program state, making the program *dependant* on their ordering. This independence relation obeys the properties explained in Section 2.2.3.1. Using Kalhon's syntax, the relation means that, should two transitions t_1 and t_2 be dependant, then we must explore both the interleaving $t_1 < t_2$ and $t_2 < t_1$ to discover all possible states. In all the examples below, we consider systems with some n threads, each with one enabled transition named t_x, where x is the thread identifier.

MPOR defines an order in which transitions are to be executed, and rules for when transitions may be executed outside of that order. Initially, transitions are only scheduled to run in the order of their thread ID numbers. For example, were we to have four threads, then we would only execute the transitions in the order of their thread IDs, i.e. 1, 2, 3, 4. No other interleavings would be explored.

When a dependency relation exists between a pair of transitions, MPOR allows the pair to swap their ordering. In these circumstances, however, there may still be a number of other transitions that can be executed independently. The thread ID ordering is still applied to all other transitions, to ensure that no spurious interleavings are then executed. Thus, were we to have four threads, with threads two and three having dependant transitions enabled, and one and four being independent, then we would explore the interleavings:

- $t_1 < t_2 < t_3 < t_4$

- $t_1 < t_3 < t_2 < t_4$

Kalhon also shows that dependency relations can exist between two transitions that do not have an explicit dependency, caused by a common intermediate dependency relation. For example taking three threads, consider the case where t_1 depends on t_2, and t_2 depends on t_3. Clearly, we must explore cases where each dependant pair of transitions are executed outside of their thread ID order, but we must also explore the interleaving $t_3 < t_2 < t_1$ as it could lead to a different state too. This circumstance is characterised by observing that two transitions between which there exists a *chain* of transitions which are pairwise dependant must also be dependant, and thus must be scheduled out of the usual order.

Finally, Kalhon identifies a pathological scheduling case that is impossible to schedule using the above approach. Consider four threads in which t_1 and t_4 are dependant, and t_2 and t_3 dependant. Here, we must explore the out of order interleavings of t_1 and t_4. However, under the ordering rules described above, this means that the prefix $t_4 < t_1$ is explored, after which we are unable to schedule either t_2 or t_3. The independent relation rule says that they must be scheduled both after t_1 but before t_4, which is impossible. To avoid this case, Kalhon weakens the scheduling constraints so that if we cannot produce an interleaving honouring the thread ID order, we restart the scheduling order after the already scheduled portion of transitions, starting from the lowest available thread ID. In the example given, this leads to the interleavings:

- $t_1 < t_2 < t_3 < t_4$
- $t_1 < t_3 < t_2 < t_4$
- $t_4 < t_1 < t_2 < t_3$
- $t_4 < t_1 < t_3 < t_2$

Which covers all discoverable behaviours. These transition scheduling rules are formalised into two definitions by Kalhon et al. [106], which I quote here:

Definition 5.1. Dependency Chain *Let t and t' be transitions executed along a computation x such that $t <_x t'$. A dependency chain along x starting at t is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ executed along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is dependent with tr_{i_j+1} , and (c) there does not exist a transition executed along x between tr_{i_j} and tr_{i_j+1} that is dependent with tr_{i_j}. A dependency chain along path x is denoted $tr \rightarrow_x tr'$.*

Definition 5.2. Quasi-Monotonic Computation *A computation x is said to be quasi-monotonic if and only if for each pair of transitions tr and tr' such that $tr' <_x tr$ we have $tid(tr') > tid(tr)$ only if either (i) $tr' \rightarrow_x tr$, or (ii) there exists a transition tr'' such that $tid(tr'') < tid(tr)$, $tr' \rightarrow_x tr''$ and $tr' <_x tr'' <_x tr$*

5.3.2 Implementation within ESBMC

Unfortunately the MPOR approach itself is not directly applicable to ESBMC. States and transitions in Kripke structures are finite and well defined, however as bounded model checking deals with potentially infinite state systems it does not enumerate all possible states, and instead repeatedly executes the transition relation to reach new states. Due to pointer indirection, it is not possible to know what variables are accessed by some transitions, meaning we are unable to identify statically what dependant transitions exist in a program.

While ESBMC encodes the logic in the program under test symbolically, the scheduling of threads occurs explicitly, creating one symbolic program formula per unique thread schedule. This is in contrast to Kalhon's experiments [106], where the ordering of transitions in a model are chosen by the solver, and constrained to honour the MPOR algorithm by the formula. In this respect, ESBMC has more in common with explicit state model checkers when performing partial order reductions (Section 2.2.3.1) than purely symbolic model checkers.

In addition, ESBMCs default mode of multithreaded operation does not aid the analysis of transitions. Rather than considering whether to take a context switch after each statement of the program, consideration only occurs after a thread accesses shared state or a synchronisation primitive. This method constitutes a crude partial order reduction in itself. It means, however, that when ESBMC begins executing a thread it does not know in advance when it will stop and consider context switching. This is incompatible with MPOR, as we must know what dependant transitions are available in advance, and then decide how to schedule them.

Given these obstacles, when implementing MPOR within ESBMC, I execute every transition that is available within ESBMC (that is, a sequence of statements accessing thread local data, up to a global variable access were we *might* context switch) from each state discovered, and then retrospectively analyse whether or not the run just taken would have been permitted by the rules governing MPOR. If it would have been permitted, then exploration continues as normal. If not, the interleaving is abandoned, we backtrack to the point where the previous context switch was taken, and a new path is taken. This has the property that, while un-necessary transitions are explored, they do not contribute to the state explosion problem, as the reachability tree from the un-necessary transition is *not* explored. In the worst case, where all running threads are entirely independent (meaning MPOR only allows a single ordering of all transitions), the number of un-necessary transitions taken are the number of threads minus one, times the number of statements where a context switch may occur. This grows linearly with the number of interleaving points and threads, rather than exponentially.

Within ESBMC, the optimisation requires the collection of certain pieces of data to calculate the dependency relations between the thread transitions taken. In the context of ANSI-C software, we say that two thread transitions are dependant if they either both write to the same shared variable, or if one writes and the other thread reads the same shared variable. The case of both threads reading a shared variable cannot lead to a different state, and thus both threads are independent. Synchronisation primitives within ESBMC are modelled as normal C operations on shared variables, and thus do not require special consideration. During the execution of a transition, the set of shared variables accessed (and the kind of access which occurred) is stored. Later, when a context switch is considered, the per-thread sets of accessed variables are compared. If a dependant pair of reads/writes is discovered between two threads, then the threads are marked as depending on each other in this particular state. This ability to classify whether the actions of threads are dependant or independent allows us to evaluate the MPOR scheduling rules.

5.3.3 Evaluation

As with state hashing, the MPOR optimisation was applied to the SV-COMP'14 benchmark suite in the manner described above. The longest running test results are shown in Tables B.4 and B.5. Most strikingly, *all* the longest-running runs have a performance improvement, the smallest improvement being a reduction in time consumed of one third. The mean reduction in performance is by 90% of the original amount of time consumed, an order of magnitudes improvement, with a median reduction of 88%.

MPOR does not result in any newly-failing runs, and a total of 295 test runs are newly succeeding.

Across all the 1973 commonly successful runs, 794 showed no change in execution time, 1057 took less time to complete when the MPOR optimisation was enabled, and just 122 tests increased their time consumption. The greatest speedup was by 10636 seconds—the best 48_ticket_lock_low_contention_vs_true.i[8] with unwind bound 2 and context bound 6 produced 1.35 million interleavings when run without optimisation in 10649 seconds, but when MPOR was enabled, it took just 13 seconds to explore 22,000 interleavings and conclude that it had explored all the necessary interleavings.

The mean performance improvement over the commonly successful runs was a reduction in time consumed by 336 seconds, while the median was 6 seconds. 273 of the improved runs showed performance gains of more than 100 seconds. 79 improved by more than 1000 seconds.

Most impressively however, all the tests that had a reduction in performance as a result of the MPOR optimisation only consumed at most two seconds more time. The penalty of using MPOR on this set of benchmarks is thus negligible.

5.3.4 Summary

The monotonic partial order reduction has demonstrated significant performance improvements over the unoptimised version of ESBMC. Not only is there an order of magnitudes improvement

[8]The same test was the most improved with state hashing; the significance of this is discussed in Section 5.7

in time consumption (on average over the longest running tests), but there are no tests where its use leads to a significant decrease in performance. The cost of using it is therefore negligible.[9] The performance improvements are not across the whole set of SV-COMP benchmarks; however for the subset of tests where a performance improvement was observed, the improvement was significant. Additional work is required however to ensure that ESBMCs current feature set still works under this mode of operation.

5.4 Incremental SMT and solver queries

The most recent revision of the SMTLIB logic format [24] (the standardised input language to SMT solvers) have introduced the concept of an *assertion stack*, and the ability to push and pop assertions on and off it. This means that all SMTLIB compliant SMT solvers must have an internal stack of assertions that the user has applied to the variables in the formula, and that in the course of interpreting the input file new assertions can be added to or old assertions removed from this stack. The desired outcome of this approach is to enable *assertion retraction* and *lemma learning*. The former allows one to speculatively add assertions to the formula, evaluate the result, then return the formula to its original form. The latter is the process whereby the SMT solver stores facts (in the form of lemmas over the formula variables) that it has determined about the formula, that may prove useful in future checks of the formula. This enables the solver to use lemmas determined during previous checks for future checks, optimising the search procedure and potentially eliminating a large amount of formula state space to be searched. The term for this technique is “Incremental SMT”, where the formula is built up in stages, and lemmas learnt about the formula along the way.

This facility is potentially beneficial to the exploration of multithreaded state space by ESBMC. The existing operation of the SMT solver follows directly from the original implementation within CBMC: once a SSA program has been produced by symbolic execution, that program is converted to first order logic and translated to a form acceptable to the solver, after which it is solved. After solving, the entire formula is discarded. While this is obviously correct for single threaded exploration, during multithreaded exploration a large number of SSA programs will be converted, solved, and discarded. Using assertion retraction to build and deconstruct the formula could allow reduced SMT-conversion overhead, and lemma learning could lead to swifter verification times. The default SMT solver in ESBMC, Z3, claims lemma learning as a feature [72], allowing us to evaluate it’s impact.

One issue of nomenclature arises—in ESBMC, an assertion is a program property that we search for violations of. However in SMT solvers, an assertion is a constraint on the variables in formula that must hold if the formula is to be satisfiable. As program properties are not immediately relevant, my use of “assertion” in this section means an assertion in the SMT solver.

[9] All future releases of ESBMC have MPOR enabled by default

5.4.1 Encoding SMT during symbolic execution

To make use of incremental SMT during multithreaded state exploration we must identify ways in which the SMT formula can be re-used between SSA programs, through pushing and popping assertions into the solver. The most obvious way of using incremental SMT is to retain the formula produced from one SSA program, identify the common prefix between it and the next SSA program produced, and retract all the assertions that are not common (i.e., that were specific to the previous interleaving). Then, place the new assertions from the next SSA program on top of the remaining formula.

To illustrate this, consider Figure 5.3. The first set of lines represent four different interleavings of a particular trace when explored normally—all the transitions are encoded, solved, and then discarded, before we move onto considering the next interleaving. The colours indicate identical transitions, i.e. transition A is common to all interleavings, while C and D are not. The tree like structure below represents the formula when using incremental SMT. The first interleaving produces a formula up to the end of Run 1, after which we backtrack through transition F and then take a different transition, E, to produce Run 2. The same approach leads to the other branch of the tree (runs 3 and 4). Observe that transitions A and B are not discarded and so only need to be encoded once, and any lemmas learnt over the assertions that make up those transitions will exist for all subsequent paths and thus help optimise them. Transitions C and D are also present for at least two of the illustrated runs, and lemmas learnt there will optimise a portion of the explored tree. Clearly, the most optimisation is achieved when as much formula as possible is preserved for as long as possible, decreasing the amount of formula that must be encoded and increasing the number of learnt lemmas.

Implementing this requires ESBMC to preserve a single SMT formula throughout state space exploration, and to retract assertions from the formula to an appropriate point, once the formula has been checked. The former can be achieved through a number of methods—the SMTLIB format itself allows input files to be treated as a script, allowing ESBMC to send commands to the solver and read back results on-the-fly (in the typical UNIX tool style of processes communicating through a stream of characters). Other SMT solvers provide a library and API for solving in-process. The latter feature, backtracking, requires ESBMC to issue SMTLIB push and pop commands; pushing before encoding assertions to the solver, and popping when backtracking. The exact points to "push" (mark a point on the assertion stack that a future "pop" will retract assertions up to) are those points in execution where context switches are taken. This means individual transitions can be popped out of the SMT formula as a whole.

This optimisation is enabled on the ESBMC command line through the option "–smt-during-symex", and is referred to from here as the incremental solving optimisation.

One unforeseen outcome is that the "slicing" optimisation inherited from CBMC becomes unviable. Slicing occurs by taking a completed SSA program and walking backwards through the program, marking variables that properties depend upon. Once completed, any variables that

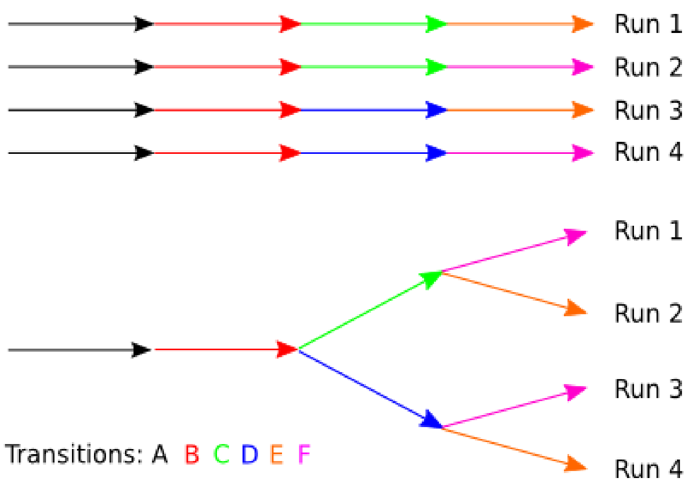

FIGURE 5.3: Example representation of the SSA programs making up a multithreaded exploration. The first set of runs illustrate the programs produced during normal exploration, while the second shows the tree-structure produced by incremental SMT

properties do not depend upon are eliminated. In incremental solving, however, we do not know what properties may be checked in a future interleaving, and so we must encode all assignments to the formula when performing incremental SMT.

5.4.2 Eliminating unfeasible paths

By keeping an SMT formula for the current program path available throughout state exploration, we create opportunities for further optimisations. We can reduce the number of interleavings we must explore by eliminating any *unviable paths*, i.e. paths that pass through an unreachable state. The reachability subtree of any unreachable state is also unreachable. ESBMC already eliminates some unreachable states by statically determining whether branch guards can be true or false, however this does not always succeed. With the SMT formula available however, we can test whether a state is unreachable during exploration.

At any point in the symbolic execution of the program, the condition for the current path being viable or not is represented by the *execution guard*, the combination of the guards on all the branches taken to reach this point. We can query the SMT solver and determine whether this execution guard can ever be true. If it can be true, then we must continue exploration of the current path. However, if the guard can never be true, then we have discovered an unreachable state, and can immediately backtrack to a reachable one, avoiding the exploration of additional interleavings.

```c
void *thread1(int lockorder) {
  pthread_mutext_t *lock1, *lock2;
  lock1 = (lockorder == 0) ? &alpha_lock : &beta_lock;
  lock2 = (lock1 == &alpha_lock) ? &beta_lock : &alpha_lock;

  pthread_mutex_lock(lock1);
  /* Perform some unknown operation */
  pthread_mutex_unlock(lock1);
}
```

```c
void *thread2(int lockorder) {
  pthread_mutext_t *lock1, *lock2;
  lock1 = (lockorder == 0) ? &alpha_lock : &beta_lock;
  lock2 = (lock1 == &alpha_lock) ? &beta_lock : &alpha_lock;

  pthread_mutex_lock(lock2);
  /* Perform some unknown operation */
  pthread_mutex_unlock(lock2);
}
```

FIGURE 5.4: Program with nondeterministic locking of a lock

To demonstrate the usefulness of this, consider the example in Figure 5.4. Here, the first thread nondeterministically locks a shared mutex, according to the condition `lockorder`. The second thread behaves in the same manner, deciding which lock to use according to the function argument. Should that argument be nondeterministic (i.e. sourced from a program input or explicit nondeterministic variable), ESBMC is unable to determine statically *which* of the two locks is being locked. As a result, numerous interleavings are produced and explored, including those where both threads have locked the same lock (an unfeasible state), to cover all possible behaviours.

Within ESBMC, I have implemented this by querying the solver whenever we reach a context switch point. An assertion that the execution guard is true is encoded, and the formula checked to see if it is satisfiable. If it is not, we know that the current path is not viable, and we can cease exploration and backtrack.

The idea of querying a theorem solver to guide the exploration of a program is not new. Existing symbolic execution tools [48] and static analysers [161] query solvers to identify unfeasible paths. I am unaware, however, of any model checkers that implement this approach (although Günther et al. propose to implement this soon in [89]), or of any tools that apply this approach to concurrent software verification.

5.4.3 Evaluation

Two potential optimisations were proposed in this section: the first is that by incrementally encoding the SMT formula during symbolic execution, we can reduce the amount of time that it takes to verify a state through reduced SMT encoding overhead and learnt lemmas. The second is that by querying the SMT solver during multithreaded state exploration as to whether the current interleaving is feasible, we can identify unfeasible states in the program early and avoid having to explore all subsequent interleavings from it.

As with the previous optimisations, these were tested against the SV-COMP'14 benchmarks in the manner described in Section 5.1.

5.4.3.1 Incremental solving

The performance impact on the longest running test runs are shown in Tables B.6 and B.7. Unfortunately, the optimisation has increased the memory consumption of some of the runs, leading to 17 of the longest running to crash in out of memory conditions. Additionally, 17 more of the runs exhibit a decrease in performance. Not all the effects are negative—the rest of the runs show some form of performance increase, although not of the same magnitude as the previous two optimisations evaluated.

On the newly-failing and newly-succeeding front, results are mixed too. 44 runs newly-fail with incremental solving (some being those presented in Table B.6), while only 12 newly succeed. Any tests newly failing as a result of an optimisation is a very bad sign.

Considering the set of commonly successful runs (1929 in total), 805 runs showed no difference between optimised and unoptimised versions. In 332 runs the optimised version of ESBMC ran faster than the normal, and in 792 runs it ran slower. This hints at a general case slowdown in performance. In terms of averages, the runs where the optimised version was slower were slower by a mean of 33 seconds, a median of 12, whereas those runs that were faster with the optimised version had an average improvement of 366 seconds, but a median of 1 second improvement. The maximum improvement was of 7901 seconds (32_pthread5_vs_false.i with unwind bound 1 and context bound 7), while the greatest decrease was 1190 seconds (07_rand_true.i with unwind bound 3 and context bound 4).

Given the median performance effect and that the majority of tests run slower with this optimisation shows it is not generally useful across the set of benchmarks, in many cases being a performance hindrance. Curiously, there is a small subset of runs where large performance improvements are seen, as shown in Table B.10. Here, the first 35 largest changes in consumed time over all commonly successful runs are all speedups as a result of the optimisation, and come to a total of 95122 fewer seconds of time consumed. The 35 runs with the largest change come from a total of twenty benchmarks. These improvements seem to indicate that there is a small subset of benchmarks where this optimisation proves useful.

My conclusion is that the incremental solving optimisation can deliver performance improvements in a small number of cases, but that it is unsuccessful as a general optimisation.

5.4.3.2 Thread guard

The thread guard optimisation requires the incremental solving option to be enabled for it to operate, and so the results below suffer some of the same performance problems that incremental solving did. I chose not to try and separate the results though, as there is no way to separate the optimisations.

There are still 14 newly-failing runs (i.e. those where the unoptimised version of ESBMC succeeds but the thread-guard optimisation fails), although 61 tests newly succeed. Again, this is not a good result.

Tables B.8 and B.9 show the performance changes on the longest running runs as a result of the thread guard optimisation. We see a number of tests newly failing (crashing) although this is fewer than with just incremental solving (5 newly failing vs 17). More importantly, however, more tests show a performance improvement, with 51 out of 67 showing a reduction in time elapsed, with a broad range of percentage changes.

Turning to the commonly successful runs, a total of 1959 runs produced results in both versions of ESBMC. 730 showed no change in time, 661 speed up with the thread-guard optimisation, and 568 slowed down. The unoptimised version of ESBMC consumed 358900 seconds across these runs, while the thread-guard optimised version consumed 111632 seconds. Those runs that sped up contributed 266882 fewer seconds of time consumption, while those that slowed down contributed an additional 19614 seconds.

The largest speedup was of 12230 seconds, 85% of the time allocation the test was permitted. The largest slowdown was of 1162 seconds. On average, the mean performance increase was 403 seconds on tests that sped up, with a median of 34 seconds. Tests that slowed down did so by a mean of 34 seconds, a median of 2 seconds.

Of these runs, 185 speed up more than 100 seconds across 45 tests, while 44 slowed down more than 100 seconds across 15 tests.

These results show that, versus an unoptimised version of ESBMC, the thread guard optimisation can deliver higher performance; however the performance gain varies significantly, depending on the nature of the test. Taking for example, the `singleton_false.i` benchmark, more than an order of magnitude speedup is observed (12562 seconds reduced to 332 seconds) with this optimisation.[10] The test itself generates four threads, each performing a trivial assignment; they are then all `pthread_join`'d[11]. The implementation of this API call simply encodes an assumption that the designated thread has completed, and continues execution. As a result, it is likely that a large number of the interleavings explored could feature a false assumption that another thread has terminated, which would be identified by the thread-guard optimisation and pruned. This is backed up by the interleaving numbers for that test: with an unwind bound of 1

[10] In this case, the incremental SMT optimisation contributes some 1500 seconds of the performance improvement

[11] A function that blocks execution until a specified thread terminates

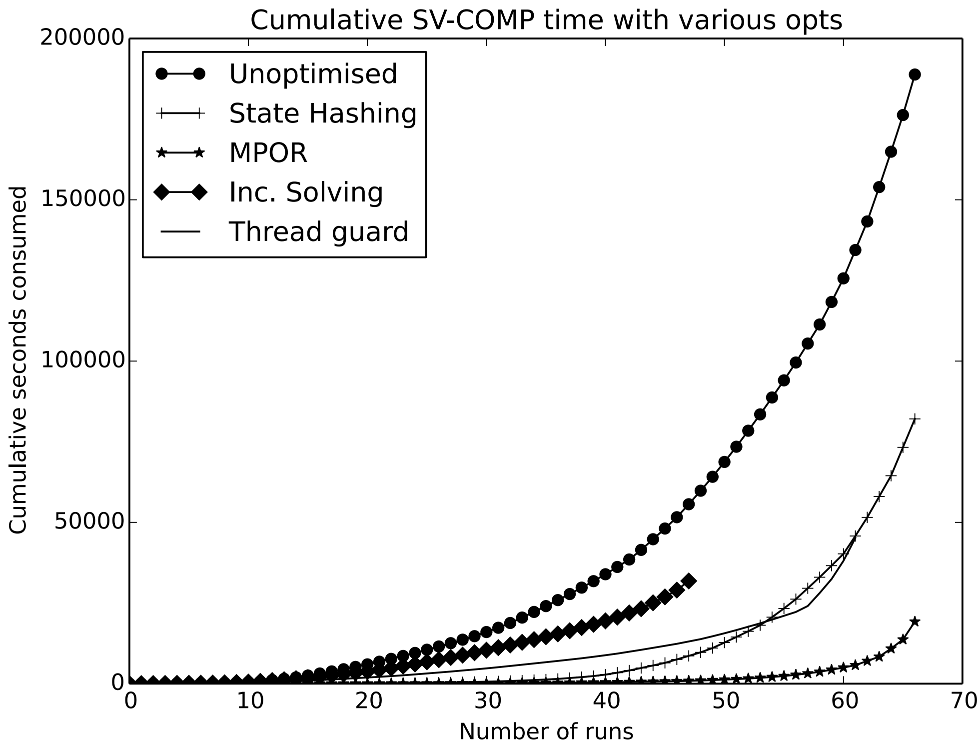

FIGURE 5.5: Cumulative sum of run time, ordered by runtime, for all optimisations discussed in this chapter versus unoptimised ESBMC

and context switch bound of 5, 1,253,886 interleavings are explored in the unoptimised version of ESBMC, and only 299,175 when the thread-guard optimisation is enabled.

My conclusion is that certain benchmarks can have a significant performance improvement as a result of this optimisation, but that it should not be considered a general purpose solution. There is a small (six) set of tests where the thread guard optimisation improves performance by more than both the state hashing and MPOR optimisations, most notably singleton_false.i and 04_incdec_cas_true.i, which should not be overlooked.

5.5 Discussion

Of the four optimisations examined, we have seen that the MPOR algorithm is the most effective at increasing the performance of model checking of concurrent software across the benchmark suite, delivering an order of magnitude improvement in speed without seriously reducing the performance of any run. This is illustrated in Figure 5.5, where I compare the performance of the different optimisations explored in this chapter. The y axis is the cumulative amount of verification time when runs are ordered by verification time. The exponential curves are thus a product of the graph format; what is of interest is the rate of increase and heights.

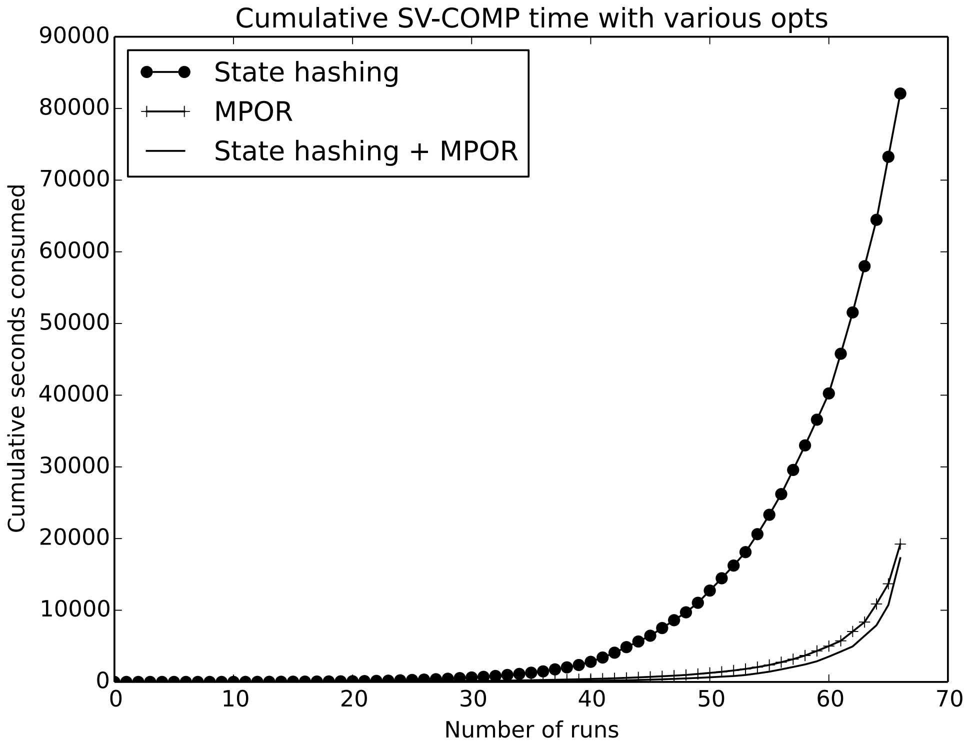

FIGURE 5.6: Cumulative sum of run time, ordered by runtime, for the MPOR and state hashing optimisations, versus the combination of the two

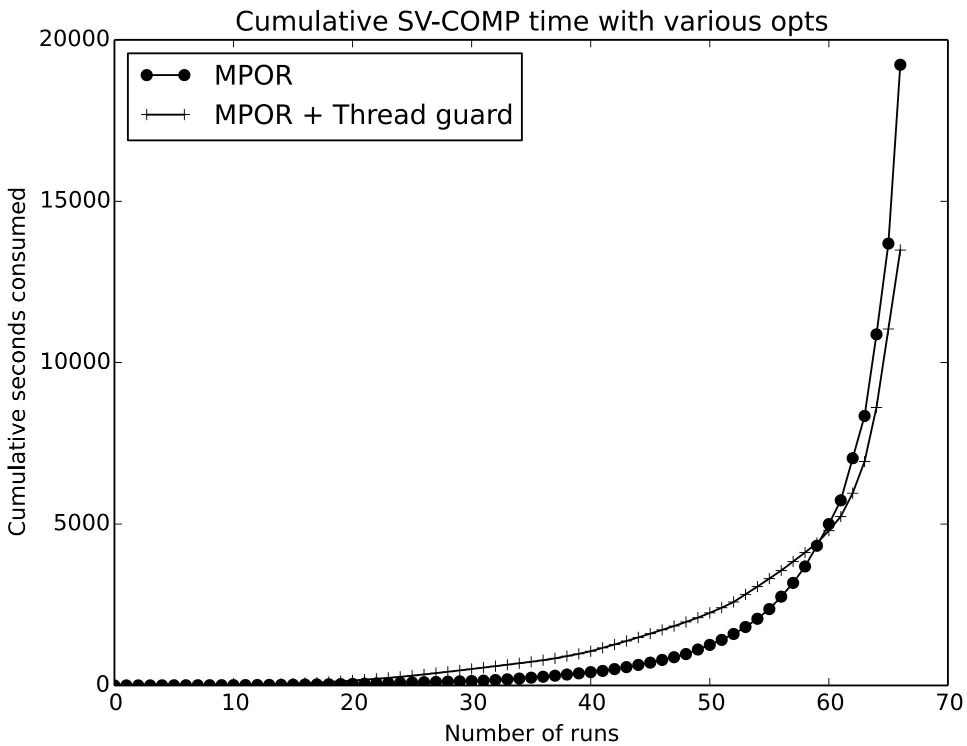

FIGURE 5.7: Cumulative sum of run time, ordered by runtime, between the MPOR optimisation and a combination of MPOR and thread guard optimisations

Observing the curve of the MPOR optimisation (stars) versus unoptimised ESBMC (circles), we see that the total cumulative verification time up to the 50th smallest test is negligible, compared to roughly 70,000 seconds of verification time without optimisation. After this point the MPOR optimisation begins accumulating verification time faster, but ends with a total of 20,000 seconds of verification time, compared to almost ten times as much in unoptimised ESBMC.

A similar progression is observed for state hashing (crosses), although it's curve begins ascending earlier and at a much faster rate than MPOR. This shows the performance benefits of state hashing to be less than those of MPOR, although still an improvement on the unoptimised version. As concluded in Section 5.2, state hashing generally leads to a performance improvement, but can sometimes result in performance decreases for a small number of benchmarks.

Both incremental solving and the thread guard optimisations have incomplete data sets, due to running out of memory. Note that the thread guard line terminates in the middle of the state hashing line, five marks from the rightmost mark. It would appear that incremental solving is not as effective as the other optimisations, and that the thread guard optimisation is almost as effective as state hashing, however it would be wrong to draw any firm conclusions without the final data points.

I now examine the effect of combining certain optimisations. The state space pruned by each optimisation (save incremental solving, which sought to improve the performance of exploring individual states) does overlap in some ways. State hashing will eliminate symmetric interleavings where different threads assign the same values to variables after a context switch, and likewise the MPOR algorithm will eliminate those interleavings where the same assignments happen, as long as there is no inter-thread dependency. The two do not optimise out the same set of states though; state hashing cannot eliminate identical states that are not exactly symmetric, and MPOR cannot eliminate interleavings where there is a cross-thread dependency but the value of the read or write does not actually change the state. This can be seen best in the run that both optimisations improved the most, 48_ticket_lock_low_contention_vs_true.i with unwind bound 2 and context bound 6. From a total of 1.3 million interleavings produced in the unoptimised version, state hashing reduced this to 71,000 while MPOR produced only 21,000.

Figure 5.6 presents the cumulative time consumed by state hashing and MPOR optimisations over the benchmarks, and the effect on performance of enabling both optimisations. With both enabled, ESBMC performs marginally better than either of the optimisations by themselves. The difference is not large, but it does demonstrate that the two optimisations are affecting different sets of interleavings.

The thread guard optimisation does not relate to either state hashing or MPOR, as it checks the in-program property of whether the path being taken is feasible. These experiments have discovered, however, that the performance to be gained from this optimisation is not as significant as the others—there are fewer newly-feasible test runs, and some test runs do not complete verification where they normally would. To better understand the performance of the thread guard optimisation, Figure 5.7 presents the cumulative time consumed over the benchmarks by MPOR, and MPOR with the thread guard optimisation. By using the MPOR optimisation to reduce the number of states to be explored, we ensure that all of the runs using the thread guard optimisation succeed, meaning we can fully compare the performance of using it. For the majority of the tests the thread-guard optimisation results in higher verification times, however towards the end of the series the lines cross, indicating that for some runs the thread guard optimisation yields a lower verification time, to the extent that across the whole set of benchmarks, thread guard delivers a slight performance improvement. This backs up the conclusion of Section 5.4.3.2, that the optimisation is effective for some verification problems (but not others).

Testing ESBMC against the benchmarks again, with all the explored optimisations enabled, 332 additional runs complete their verification within the resource limits given versus an unoptimised version, versus a single test (07_rand_true.i) that no longer finishes successfully. This is a useful outcome from these optimisations.

5.6 Related work

Concurrent software verification is a mature field with a large variety of research currently taking place. I am not aware of any symbolic model checker, or indeed software model checker, that makes use of state hashing aside from ESBMC.

The closest tools to ESBMC that use partial order reductions are CBMC [53] and JPF [157]. CBMC encodes all thread executions into one SAT formula that directs the solver to re-order their executions to explore the state space, but also adds partial order constraints to avoid redundant interleavings being considered [11]. JPF is an explicit state model checker, and uses partial order techniques to identify which states it should backtrack to during execution, depending on the interactions between threads [133]. I am not aware of any tools that apply the monotonic partial order reduction [106] to software model checking. Noonan et al. provide another optimal POR [10] that they apply to the verification of Erlang programs, and dismiss Kalhon et al. as not being applicable to stateless model checking, where state sequences are enumerated during exploration. The speculative executions and their retrospective analysis described in Section 5.3.2 provides a bridge between Kalhons work and stateless model checking.

Many other tools use partial order reductions in settings other than model checking, for example in test generation tools [105], or in simulation of other concurrent systems such as SystemC simulation [50].

Another branch of concurrent model checking is where, rather than explicitly exploring all possible interleavings in the program under test, the program is instead translated into a series of transition relations in an SMT solver and the solver is instructed to find an ordering of them that satisfies a condition (Much like the STORM encoding in Section 2.2.2). CBMC takes such an approach [12], encoding the bounded runs of each thread in a program to a SAT formula, then directing the solver to interleave them. CSeq [101] takes a different approach and reduces concurrent programs to a single threaded program, with a driving loop that nondeterministically executes portions of threads, allowing the solver to decide which portion of a thread is executed, when. All of these tools have shown significantly better performance then ESBMC in many benchmarks. I believe that ESBMCs explicit exploration of interleavings may still have its place, however, as it is easily distributed across a compute cluster and will consistently result in smaller SMT formulae.

5.7 Conclusions and future work

This chapter has explored four potential optimisations for reducing the amount time required to perform verification of concurrent software in ESBMC. Applying each benchmark individually to the SV-COMP'14 benchmark suite for concurrent programs, I have shown that both state hashing and monotonic partial order reduction lead to more completed test runs within the given time and memory bounds (while not making additional runs fail), and that across the range of benchmarks within the suite, their benefit outweighs the impact on tests that show a reduction in performance. The monotonic partial order reduction is clearly the better of the two, having a negligible performance penalty and delivering significant improvements.

I have also shown that there are few benefits to be had from encoding the SMT formula during state exploration, with the incremental solving optimisation. However there is also evidence that the optimisation enabled by it, of testing the feasibility of an interleaving during exploration (the "thread guard" optimisation), has some performance benefits too.

Overall, I have evaluated these optimisations against the chosen benchmark suite and discovered the different strengths of the optimisations, which have not been applied to symbolic software model checkers before, to the best of my knowledge.

For future work, it is likely that some adjustment of how frequently the thread-guard optimisation is evaluated may yield better results. Solving a small set of interleavings at the same time using the same solver context may also lead to a performance improvement, although this is distinct from the solver-scheduled SMT encodings mentioned above. Finally, the recent standardisation of C11[7] with a weak memory model opens new doors for software verification, as memory model optimisations can now be applied to all C11 code. Additionally, it defines new undefined behaviours (i.e., program properties that can be violated) that may allow model checkers to reject concurrent behaviour that exhibits races early, rather than having to find a path to a program property that can be violated as a result.

Chapter 6

Conclusions and future work

This thesis has examined the state of the art in symbolic software model checking for concurrent programs and checking of LTL formulae over software. The state space explosion problem is identified as an ever-present limitation on the feasibility of model checking in general, along with a general lack of LTL model checking techniques for software, although several methods extract models from software and then verify the model [94, 80].

To address the matter of LTL formulae, we have taken the usual technique of evaluating Büchi Automata over system states, but have synthesised the model to an ANSI-C implementation and composed it with the program under test, leading to a symbolic evaluation of the BA. We then extended the truth domain of LTL to represent the different circumstances that a finite prefix of an infinite trace may terminate in. We test this approach on a toy concurrent program, a piece of industrial software, and a suite of synthesised benchmarks and find it to be an effective approach: with the caveat however, that the unfair scheduling of threads makes the checking of liveness properties in concurrent programs unfeasible.

Once the performance limitations of our initial approach were addressed, the performance implications of our technique are small. Compared with verifying a reachability property using BMC, our technique effectively only requires an inlined function to be run whenever certain global variables are modified. This overhead grows linearly with the number of global variable modifications in the program, although the logic of the inlined BA must still be explored by the SMT solver. This means that LTL model checking can now be used anywhere that BMC is used today. There is still the finite trace limitation, in that certain properties such as true liveness will never yield a conclusive answer. However an engineer equipped with our property analysis will still be able to identify behaviours that likely violate the property.

We then turn to the performance of SMT solvers when evaluating the formulae produced by ESBMC. We select the most commonly used set of SMT solvers in the field of software verification, implement support for them in ESBMC, and compare their performance over a benchmark suite. Boolector is identified as being the fastest of the solvers, in line with expectations. We then identify two adjustments to ESBMCs SMT encoding that can be made, that both lead to reasonable improvements in performance as measured by number of tests completing verification. None of the solvers or improvements amount to an order of magnitudes performance increase, however this is no surprise as the SAT problem has been shown to be NP-complete, and so will always remain a difficult task to solve. I have shown however that there are performance enhancements to be made in how the model checking problem is encoded, and that implementing such optimisations allows us to verify a larger set of benchmarks.

Finally, a number of optimisations are identified to improve the speed of concurrent program verification, through reducing the number of interleavings to be explored and increasing the speed at which each interleaving is checked. Each is evaluated on a benchmark suite, and finally compared, with the monotonic partial order reduction (MPOR) being identified as the most effective optimisation, which is complimented to some extent by the use of state hashing. The MPOR optimisation demonstrates on average an order of magnitudes increase in performance and negligible other performance costs, making it well suited for future concurrent software verification. The other optimisations examined demonstrate performance improvements but occasionally performance costs on some benchmarks. These optimisations need not be ignored, however, as they can be made available to engineers as options, and can be assessed as to whether they are appropriate in particular use cases.

6.1 Main contributions

This thesis contains two main contributions: first and foremost, a technique for model checking LTL properties over bounded symbolic traces of software, which has not previously been achieved in the literature. This technique has been evaluated over several pieces of software, including a benchmark suite designed for LTL model checking tools, and has shown to give correct and informative results for all.

The second contribution is the evaluation of a set of optimisations to tackle the state explosion problem, both for concurrent software and to improve the normal solving of sequential program traces. Some of these techniques are completely novel, while others have not been applied to either symbolic model checking or even software model checking, before. The evaluation has shown that many of these optimisations are suitable in all available benchmarks, and lead to an increase in the range of programs that can be feasibly model checked within certain performance bounds.

6.2 Future work

The technique demonstrated in this thesis for the verification of LTL properties over software opens the door for substantially more expressive properties to be verified in software. The actual implications of this can only be explored with an industrial case study, or other evaluation of the technique against a real world piece of software, ideally as it is developed. Of particular interest is the implications of the bounding of inner loops: as discussed in Section 3.4.1 the finite prefixes we explore are not complete or possibly well defined if loops other than the outermost are bounded. Embedded software on microcontrollers tends to implement a state machine with an infinite outer loop, making such applications an appropriate study topic.

It would also be worthwhile to study precisely how constructs in software translate to complexity in SMT formula. Specifically, the ability to quantify the cost of particular expressions or estimate the additional cost of extending the unwind bound further would help the verification engineer understand what is feasibly verifiable. There is also scope for such information being used to either guide abstractions or optimisations of the formula encoding, with a view for further performance enhancements.

Finally, there are numerous engineering and maintenance tasks to perform within ESBMC that could lead to better verification in the future. Recent work [140] has extended ESBMC to support the C++ language, however supporting the entire standard within a single verification tool is unfeasible. Instead, converting the frontend of ESBMC to use the Clang compiler library would be a much more sustainable approach, and without loss of precision as clang exports it's abstract syntax tree to all library clients.

6.3 Concluding remarks

The work I have contributed here extends the bounds of the properties and programs that can be verified by model checking tools—however it is no panacea, and there are many further challenges that the model checking community must face. In particular, the state space explosion problem is never likely to be truly defeated bar some incredible breakthrough. Instead, the model checking community must go on proposing and evaluating small improvements to the techniques already present in the literature, and most importantly sharing their ideas and implementations. This is the best approach to truly increase the size of program that can be feasibly verified by automated tools, and will keep the field interesting for years to come.

Appendix A

Code samples

A.1 Sample monitors

```
char __ESBMC_property___cexpr_0[] = "pressed";
bool __cexpr_0_status;
char __ESBMC_property___cexpr_1[] = "charge > min";
bool __cexpr_1_status;

typedef enum {T0_init, accept_S2 } ltl2ba_state;
ltl2ba_state state = T0_init;
unsigned int __visited_states[2];
unsigned int __transitions_seen;
extern unsigned int __transitions_count;

void ltl2ba_fsm(bool state_stats) {
  unsigned int choice;
  while (1) {
    choice = nondet_uint();
    /* Force a context switch */
    __ESBMC_yield();
    __ESBMC_atomic_begin();
    __ESBMC_assume(__transition_count <=
                    __transitions_seen + 1);
    __transitions_seen = __transition_count;
    switch(state) {
    case T0_init:
      if (choice == 0) {
        __ESBMC_assume((1));
        state = T0_init;
      } else if (choice == 1) {
        __ESBMC_assume((!__cexpr_1_status &&
                        __cexpr_0_status));
        state = accept_S2;
      } else {
        __ESBMC_assume(0);
```

```
      }
      break;
    case accept_S2:
      if (choice == 0) {
        __ESBMC_assume((!__cexpr_1_status));
        state = accept_S2;
      } else {
        __ESBMC_assume(0);
      }
      break;
    }
    if (state_stats)
      __visited_states[state]++;
    __ESBMC_atomic_end();
  }
  return;
}

int
ltl2ba_thread(int *dummy)
{

        ltl2ba_fsm();
        return 0;
}
void

ltl2ba_start_monitor(void)
{
        pthread_t t;

        _ltl2ba_thread_done = 0;
        _ltl2ba_state = T0_init;

        _ltl2ba_transitions_seen = _ltl2ba_transition_count;
        pthread_create(&t, NULL, ltl2ba_thread, NULL);
        __ESBMC_yield();

        return;
}

void
ltl2ba_finish_monitor(void)
{

        __ESBMC_assume(_ltl2ba_thread_done == 1);
        ltl2ba_fsm();
        assert((_ltl2ba_state != accept_S2) && ("LTL property violated"));
        return;
}
```

LISTING A.1: Initial C implementation of the Büchi automaton for the formula `!G({pressed} -> F {charge > min})` and associated helper functions.

```
char __ESBMC_property___cexpr_0[] = "pressed";
bool __cexpr_0_status;
char __ESBMC_property___cexpr_1[] = "charge > min";
bool __cexpr_1_status;

typedef enum {_ltl2ba_state_0,_ltl2ba_state_1} _ltl2ba_state;
_ltl2ba_state _ltl2ba_statevar =_ltl2ba_state_0;

void *ltl2ba_fsm(void  *d) {
  unsigned int choice;
  while (1) {
    choice = nondet_uint();
    __ESBMC_atomic_begin();
    switch(_ltl2ba_statevar) {
    case _ltl2ba_state_0:
      if (choice == 0) {
        __ESBMC_assume(1);
        _ltl2ba_statevar = _ltl2ba_state_0;
      } else if (choice == 1) {
        __ESBMC_assume(!_ltl2ba_cexpr_1_status &&
                                 _ltl2ba_cexpr_0_status);
        _ltl2ba_statevar = _ltl2ba_state_1;
      } else {
        __ESBMC_assume(0);
      }
      break;
    case _ltl2ba_state_1:
      if (choice == 0) {
        __ESBMC_assume(!_ltl2ba_cexpr_1_status);
        _ltl2ba_statevar = _ltl2ba_state_1;
      } else {
        __ESBMC_assume(0);
      }
      break;
    }
    __ESBMC_atomic_end();
    __ESBMC_switch_from_monitor();
  }
}

void ltl2ba_start_monitor(void) {
  pthread_t t;
  __ESBMC_atomic_begin();
  pthread_create(&t, NULL, ltl2ba_fsm, NULL);
  __ESBMC_register_monitor(t);
  __ESBMC_atomic_end();
  __ESBMC_switch_to_monitor();
```

```
}

_Bool _ltl2ba_stutter_accept_table[4][2] = {
{false,true}, {false,false}, {true,true}, {false,true}
};

_Bool _ltl2ba_good_prefix_excluded_states[2] =
{ true, true };

_Bool _ltl2ba_bad_prefix_states[2] =
{ false, false };

unsigned int _ltl2ba_sym_to_idx(void) {
  unsigned int idx = 0;
  idx |= (_ltl2ba_cexpr_1_status) ? 1 : 0;
  idx |= (_ltl2ba_cexpr_0_status) ? 2 : 0;
  return idx;
}

void ltl2ba_finish_monitor(void) {
  __ESBMC_kill_monitor();

  _Bool in_bad_state =
    _ltl2ba_bad_prefix_states[_ltl2ba_statevar];
  __ESBMC_assert(!in_bad_state,"LTL_BAD");

  unsigned int cursym = _ltl2ba_sym_to_idx();
  _Bool in_accept_state =
    _ltl2ba_stutter_accept_table[cursym][_ltl2ba_statevar];
  __ESBMC_assert(!in_accept_state,"LTL_FAILING");

  _Bool not_in_good_prefix =
    _ltl2ba_good_prefix_excluded_states[_ltl2ba_statevar];
  __ESBMC_assert(!not_in_good_prefix,"LTL_SUCCEEDING");

  return;
}
```

LISTING A.2: C implementation of the Büchi automaton for the formula $\neg \mathsf{G}(\{\texttt{pressed}\} \implies \mathsf{F}\{\texttt{charge > min}\})$.

A.2 Pulse Oximeter source code

This section contains the Pulse Oximeter software used in Chapter 3 for testing the LTL verification technique. It was written by my colleague Lucas Cordeiro, who also holds the copyright. I list here first the original source files using during verification, and then later the test harnesses and code errors inserted during my evaluation. The Pulse Oximeter was originally verified in [69].

The source tree is as follows: Thee first three files in the “apps” directory contains the main event loop and setup code for embedded environment. The next 13 files, in the “drivers” directory, contain software to control the peripherals of the microcontroller, including some inline assembly. This assembly was not verified by our approach, but is included here for completeness. The final two files contain a ringbuffer for storing logged messages, that are communicated out of the device via a serial link.

Note that throughout the code base, device specific constructs are guarded with “#if TARGET”, and are replaced with code guarded with “#if VERIFICATION” while verification is in progress. It is also important to note that this purely because ESBMC does not precisely model the target device, leading to syntax errors when accessing registers, rather than any more fundamental limitation.

```
/*********************************************************
* File:         main.c
* Abstract:     Implementation of the Main routine
* Platform:     AT89S8252
* Project:      Pulse Oximeter
* Author(s):    Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*********************************************************/

/***********************************************
* INCLUDE FILES                                *
***********************************************/
#include "menu_app.h"
#include "../drivers/global.h"
#include "../drivers/lcd_driver.h"
#include "../drivers/sensor.h"
#include "../drivers/serial.h"
#include "../drivers/timer.h"
#include "../drivers/keyboard.h"

/*****************************************************************
* STATIC FUNCTION PROTOTYPES                                     *
*****************************************************************/
static Data8 checkError(void);

/*****************************************************************
* STATIC DATA                                                    *
*****************************************************************/
static uData8 amount;
static char sensorVal[10], tmpp[10];

/******************************************************************************
* FUNCTION IMPLEMENTATION                                                     *
******************************************************************************/
/**
 * @brief check if there is error in data acquisition
 *
 * @retval This function returns -1 if the OEM III module
 *   is either out of track or the sensor is disconnected
 *   from the OEM III module.
 */
Data8 checkError(void){

    Data8 err=0;

#if (TARGET)

   lcd_clean();

        if(IsSensorDisconnected()){
           lcd_printf("No sensor",LINE1,1);
           err=-1;

        }
        else if(IsOutofTrack()){
           lcd_printf("OutOfTrack",LINE1,1);
           err=-1;
    }
```

```
#endif

    return err;

}

/**
 * @brief show sensor data. This should be implemented when
 *   the sensor data must be shown. This function
 *   is called every second by the sensor driver.
 */
void printValue(void){

    uData8 sensorDat;

    if(checkError()==0 && getButtonState()==FALSE){

        lcd_clean();

        sensorDat = getSenPos();

#if (TARGET)
        switch(sensorDat){

            case HR:
                sprintf(sensorVal, "HR:%d", getHR());
                    lcd_printf(sensorVal,LINE1,1);
                break;
            case SPO2:
                sprintf(sensorVal, "SPO2:%d",getSpO2());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case SPO2D:
                sprintf(sensorVal, "SPO2D:%d",getSpO2D());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case SPO2FAST:
                sprintf(sensorVal, "SPO2F:%d",getSpO2Fast());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case SPO2B:
                sprintf(sensorVal, "SPO2B:%d",getSpO2B());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case EHR:
                sprintf(sensorVal, "EHR:%d",getEHR());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case ESPO2:
                sprintf(sensorVal, "ESPO2:%d",getESpO2());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case HRD:
                sprintf(sensorVal, "HRD:%d",getHRD());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case EHRD:
                sprintf(sensorVal, "EHRD:%d",getEHRD());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case ESPO2D:
                sprintf(sensorVal, "ESpO2D:%d",getESpO2D());
                lcd_printf(sensorVal,LINE1,1);
                break;
            case SREV:
                sprintf(sensorVal, "SREV:%d",getSREV());
                lcd_printf(sensorVal,LINE1,1);
                break;
        }
#endif
    }
}


int main (void){

#if (TARGET)

        lcd_clean();          /* clean display 16x2 */
```

```
        serial_init(9600);    /* configure the serial port */
        initTimer0ms(600);
        initTimer0s(1);
        initSensor();
        initMenuapp();
        initLog(2)
        insertLogElement(2);
        insertLogElement(2);
        insertLogElement(2);
        while(TRUE){

                /* Infinite loop */

        }

#endif

}
```

LISTING A.3: Source code to Pulse Oximeter file pulse/apps/main.c

```
/*********************************************************
* File:        menu_app.c
* Abstract:    Implementation of the application menu
* Platform:    AT89S8252
* Project:     Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*********************************************************/

/***********************************************
* INCLUDE FILES                                *
***********************************************/
#include <assert.h>
#include "menu_app.h"

/***********************************************
* LOCAL MACROS                                 *
***********************************************/
#define BOUNCEVAL       (2)
#define PROGRESS                (100)

#define SETMSTATE(ST)           \
        if (bounce==0) {        \
            mstate=ST;          \
       bounce=BOUNCEVAL;    \
    } else {                \
        bounce--;           \
        if (bounce<0) {     \
                bounce=0;       \
        }                   \
    }                       \

#define SETINC(parm)        \
    if (bounce==0) {        \
        parm++;             \
        bounce=BOUNCEVAL;   \
    } else {                \
        bounce--;           \
        if (bounce<0) {     \
            bounce=0;       \
        }                                       \
    }                       \

#define SETNEG(parm)            \
    if (bounce==0) {        \
        parm=!parm;         \
        bounce=BOUNCEVAL;   \
    } else {                \
        bounce--;           \
        if (bounce<0) {     \
            bounce=0;       \
        }                   \
    }                                           \

#define SETDEC(parm)        \
    if (bounce==0) {        \
        if (parm>0) {       \
            parm--;         \
```

```
        }                   \
        bounce=BOUNCEVAL;   \
    } else {                \
            bounce--;           \
        if (bounce<0) {     \
                bounce=0;       \
        }                   \
        }                                           \

#define SETBNEG(parm)       \
        if (bounce==0) {        \
            parm=!parm;         \
       bounce=BOUNCEVAL;    \
        } else {                \
        bounce--;           \
        if (bounce<0) {     \
                bounce=0;       \
        }                   \
    }                                           \

/*****************************************************************
* STATIC FUNCTION PROTOTYPES                                    *
******************************************************************/
static void startApp(void);
static void empty(void);
static void stopApp(void);
static void setSampleTime(void);
static void enableLog(void);
static void selectHR(void);
static void selectSPO2(void);
static void setLog2PC(void);
static void connectCable(void);

/*****************************************************************
* LOCAL STRUCTS                                                 *
******************************************************************/
typedef struct {

        uData8 hr;
        uData8 spo2;
        uData8 spo2d;
        uData8 spo2fast;
        uData8 spo2b;
        uData8 ehr;
        uData8 espo2;
        uData8 hrd;
        uData8 ehrd;
        uData8 espo2d;

} showData;

/*****************************************************************
* STATIC DATA                                                   *
******************************************************************/
static uData8 stime, elog, bounce, exists_log;
static uData8 count_pos, count_elem, count, global_progress, unit_progress;
static uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
showData show;
static char menuVal[10], opData[AMOUNTOFDATA];

#include <stdbool.h>

bool p_startButton, q_startPressed = false;

/******************************************************************************
* FUNCTION IMPLEMENTATION                                                     *
*******************************************************************************/
void initMenuApp (void) {

        mstate=1;
        amount=0;
        stime=1;
        bounce=0;
        elog=TRUE;
        show.hr=TRUE;
        show.spo2=TRUE;
        show.spo2d=FALSE;
        show.spo2fast=FALSE;
        show.spo2b=FALSE;
        show.ehr=FALSE;
```

```
        show.espo2=FALSE;
        show.hrd=FALSE;
        show.ehrd=FALSE;
        show.espo2d=FALSE;
        enable_buttons=TRUE;
        exists_log=FALSE;
        log2pc=FALSE;
        connect_cable=FALSE;
        count_pos=0;
        count_elem=0;
        strcpy(menuVal, "");
        strcpy(opData, "");
        opData[0]=HR;
        opData[1]=SPO2;
        count=0;
        unit_progress=0;
}

uData8 selectItem(void) {

        if (enable_buttons) {
                switch(mstate) {
                        case SETSAMPLETIME:
                SETMSTATE(SETLOG);
                break;
                        case SETLOG:
                                if (exists_log) {
                        SETMSTATE(TRANSFERLOG);
                                } else {
                                        SETMSTATE(SETHR);
                                }
                break;
                        case TRANSFERLOG:
                                SETMSTATE(SETHR);
                                break;
                        case SETHR:
                SETMSTATE(SETSPO2);
                break;
                        case SETSPO2:
                SETMSTATE(SETHRD);
                break;
                        case SETHRD:
                                SETMSTATE(SETEHRD);
                                break;
                        case SETEHRD:
                                SETMSTATE(SETEHR);
                                break;
                        case SETEHR:
                                SETMSTATE(SETSPO2D);
                                break;
                        case SETSPO2D:
                                SETMSTATE(SETSPO2FAST);
                                break;
                        case SETSPO2FAST:
                                SETMSTATE(SETSPO2B);
                                break;
                        case SETSPO2B:
                                SETMSTATE(SETESPO2);
                                break;
                        case SETESPO2:
                                SETMSTATE(SETESPO2D);
                                break;
                        case SETESPO2D:
                                SETMSTATE(SETSAMPLETIME);
                                break;
                }

#if VERIFICATION
        assert(mstate>=SETSAMPLETIME);
#endif


    }
        return mstate;
}

static void startApp(void) {

        if (exists_log) {
                mstate=CONCABLE;
                if (connect_cable){
```

```
                        sendLog2PC();
#if (TARGET)
                        lcd_clean();
                        lcd_printf("Transferring...", 1, 1);
#endif
                }

        } else {
        enable_buttons=FALSE;
        setCountElem();
        }
}

static void empty(void) {

#if (TARGET)
        lcd_clean();
        lcd_printf("EMPTY", 1, 1);
#endif
}

static void stopApp(void) {

        if (getBufferSize()!=0){
                exists_log=TRUE;
        }
        enable_buttons=TRUE;
        mstate=SETSAMPLETIME;
}

uData8 KeyUp(void) {

        uData8 result=0;

    if (enable_buttons) {

        switch(mstate) {
            case SETSAMPLETIME:
                SETINC(stime);
                                result=stime;
                break;
            case SETLOG:
                SETNEG(elog);
                                result=elog;
                break;
                        case TRANSFERLOG:
                                SETNEG(log2pc);
                                result=log2pc;
                                break;
            case SETSPO2:
                                SETNEG(show.spo2);
                                result=show.spo2;
                break;
            case SETHR:
                SETNEG(show.hr);
                                result=show.hr;
                break;
                        case CONCABLE:
                                SETNEG(connect_cable);
                                result=connect_cable;
                                break;
                        case SETHRD:
                SETNEG(show.hrd);
                                result=show.hrd;
                break;
                        case SETEHRD:
                SETNEG(show.ehrd);
                                result=show.ehrd;
                break;
                        case SETEHR:
                    SETNEG(show.ehr);
                                result=show.ehr;
                break;
                        case SETSPO2D:
                    SETNEG(show.spo2d);
                                result=show.spo2d;
                break;
                        case SETSPO2FAST:
                    SETNEG(show.spo2fast);
                                result=show.spo2fast;
```

```
                break;
                        case SETSPO2B:
                    SETNEG(show.spo2b);
                                result=show.spo2b;
                break;
                        case SETESPO2:
                    SETNEG(show.espo2);
                                result=show.espo2;
                break;
                        case SETESPO2D:
                    SETNEG(show.espo2d);
                                result=show.espo2d;
                break;
                        default:
                                result=-1;
        }
    }

#if VERIFICATION
        assert(result>=0);
#endif

        return result;
}

uData8 KeyDown(void) {

        uData8 result=0;

    if (enable_buttons) {

        switch(mstate) {
            case SETSAMPLETIME:
                SETDEC(stime);
                                result=stime;
                break;
            case SETLOG:
                SETBNEG(elog);
                                result=elog;
                break;
                        case TRANSFERLOG:
                                SETNEG(log2pc);
                                result=log2pc;
                                break;
            case SETSPO2:
                                SETBNEG(show.spo2);
                                result=show.spo2;
                break;
            case SETHR:
                SETBNEG(show.hr);
                                result=show.hr;
                break;
                        case CONCABLE:
                                SETBNEG(connect_cable);
                                result=connect_cable;
                        case SETHRD:
                SETBNEG(show.hrd);
                                result=show.hrd;
                break;
                        case SETEHRD:
                SETBNEG(show.ehrd);
                                result=show.ehrd;
                break;
                        case SETEHR:
                    SETBNEG(show.ehr);
                                result=show.ehr;
                break;
                        case SETSPO2D:
                    SETBNEG(show.spo2d);
                                result=show.spo2d;
                break;
                        case SETSPO2FAST:
                    SETBNEG(show.spo2fast);
                                result=show.spo2fast;
                break;
                        case SETSPO2B:
                    SETBNEG(show.spo2b);
                                result=show.spo2b;
                break;
                        case SETESPO2:
```

```
                    SETBNEG(show.espo2);
                                result=show.espo2;
                break;
                        case SETESPO2D:
                    SETBNEG(show.espo2d);
                                result=show.espo2d;
                break;
                        default:
                                result=-1;
        }
    }

#if VERIFICATION
        assert(result>=0);
#endif

        return result;
}

#if 1
Data8 chooseSensorData(uData8 op, uData8 en) {

        Data8 err=0;

        if (op>AMOUNTOFDATA || op<1) {
                err=-1;
                return err;
    }

        if(en){
                opData[op-1] = op;
        } else {
                opData[op-1] = en;
        }
        err = opData[op-1];

#if VERIFICATION
        assert(err>=0);
#endif

        return err;
}
#endif

static void setSampleTime(void) {

    sprintf(menuVal, "Sample time: %d", stime);

#if (TARGET)
    lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif

}

static void connectCable(void) {


    if (connect_cable == TRUE) {
        sprintf(menuVal, "Cable? yes");
    } else {
        sprintf(menuVal, "Cable? no");
    }

#if (TARGET)
    lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif

}

static void enableLog(void) {


    if (elog == TRUE) {
        sprintf(menuVal, "Enable log: yes");
    } else {
        sprintf(menuVal, "Enable log: no");
    }

```

```
#if (TARGET)
    lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif

}

static void selectHR(void) {

    uData8 ret;

        ret = chooseSensorData(HR, show.hr);
    if (ret == 0) {
        if (show.hr==TRUE) {
            sprintf(menuVal, "Show HR: yes");
        } else {
            sprintf(menuVal, "Show HR: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectSPO2(void) {

    uData8 ret;

    ret = chooseSensorData(SPO2, show.spo2);
    if (ret == 0) {
                if (show.spo2==TRUE) {
            sprintf(menuVal, "Show SPO2: yes");
        } else {
            sprintf(menuVal, "Show SPO2: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectHRD(void) {

    uData8 ret;

    ret = chooseSensorData(HRD, show.hrd);
    if (ret == 0) {
                if (show.hrd==TRUE) {
            sprintf(menuVal, "Show HRD: yes");
        } else {
            sprintf(menuVal, "Show HRD: no");
        }
#if (TARGET)
        lcd_clean();
        lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectEHR(void) {

    uData8 ret;

    ret = chooseSensorData(EHR, show.ehr);
    if (ret == 0) {
                if (show.ehr==TRUE) {
            sprintf(menuVal, "Show EHR: yes");
        } else {
            sprintf(menuVal, "Show EHR: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectSPO2D(void) {
```

```

    uData8 ret;

    ret = chooseSensorData(SPO2D, show.spo2d);
    if (ret == 0) {
                if (show.spo2d==TRUE) {
            sprintf(menuVal, "Show SPO2D: yes");
        } else {
            sprintf(menuVal, "Show SPO2D: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectSPO2FAST(void) {

    uData8 ret;

    ret = chooseSensorData(SPO2FAST, show.spo2fast);
    if (ret == 0) {
                if (show.spo2fast==TRUE) {
            sprintf(menuVal, "Show SPO2F: yes");
        } else {
            sprintf(menuVal, "Show SPO2F: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectSPO2B(void) {

    uData8 ret;

    ret = chooseSensorData(SPO2B, show.spo2b);
    if (ret == 0) {
                if (show.spo2b==TRUE) {
            sprintf(menuVal, "Show SPO2B: yes");
        } else {
            sprintf(menuVal, "Show SPO2B: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectESPO2(void) {

    uData8 ret;

    ret = chooseSensorData(ESPO2, show.espo2);
    if (ret == 0) {
                if (show.espo2==TRUE) {
            sprintf(menuVal, "Show ESPO2: yes");
        } else {
            sprintf(menuVal, "Show ESPO2: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void selectESPO2D(void) {

    uData8 ret;

    ret = chooseSensorData(ESPO2D, show.espo2d);
    if (ret == 0) {
                if (show.espo2d==TRUE) {
            sprintf(menuVal, "Show ESPO2D: yes");
        } else {
```

```
            sprintf(menuVal, "Show ESPO2D: no");
        }
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif
    }
}

static void setLog2PC(void) {

    if (exists_log) {
        if (log2pc == TRUE) {
                sprintf(menuVal, "Send log: yes");
        } else {
                sprintf(menuVal, "Send log: no");
        }
        }

#if (TARGET)
    lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif

}

/**
 * @brief Timer interrupt
 *
 * @retval void
 *
 * Comments:
 * This procedure should be implemented if the
 * timer is configured. The code inside this
 * this procedure will be called according to the
 * time specified in initTimer0s(uData8 time_s).
 */
void timers_interrupt(void){

    if (enable_buttons) {

        switch(mstate){
                case SETSAMPLETIME:
                        setSampleTime();
                                break;
                    case SETLOG:
                        enableLog();
                                break;
                        case TRANSFERLOG:
                                setLog2PC();
                                break;
                    case SETHR:
                                selectHR();
                                amount++;
                                break;
                    case SETSPO2:
                                selectSPO2();
                                amount++;
                                break;
                    case SETHRD:
                                selectHRD();
                                amount++;
                                break;
                    case SETEHR:
                                selectEHR();
                                amount++;
                                break;
                    case SETSPO2D:
                                selectSPO2D();
                                amount++;
                                break;
                    case SETSPO2FAST:
                                selectSPO2FAST();
                                amount++;
                                break;
                    case SETSPO2B:
                                selectSPO2B();
                                amount++;
                                break;
                    case SETESPO2:
```

```
                                selectESPO2();
                                amount++;
                                break;
                    case SETESPO2D:
                                selectESPO2D();
                                amount++;
                                break;
                        case CONCABLE:
                                connectCable();
                                break;
           }
    }
}

#if 1
uData8 calculateUnitProgress(void){

        uData8 length;

        length = getBufferSize();
        if (length!=0) {
                unit_progress = (PROGRESS/length);
#if VERIFICATION
        assert(unit_progress>=0);
#endif
        }

        return unit_progress;
}
#endif

uData8 logTransferProgress(void) {

        uData8 global_progress;

        count++;
        global_progress = (unit_progress*count);
        sprintf(menuVal, "Progress: %d%", global_progress);
#if (TARGET)
        lcd_clean();
    lcd_printf(menuVal, LINE1, 1);
#endif

        return global_progress;
}

void timerms_interrupt(void){

    uData8 keys=0x00; /* no key pressed */

#if (TARGET)
    keys=P1;
    pressed_key = checkPressedButton(keys);
#endif

#if VERIFICATION
        pressed_key = startButton;
        //this indicates that startButton has been pressed
        p_startButton = 1;
        //assert(keys>=0);
#endif

    if(pressed_key>0){

        switch(pressed_key){
                case startButton:
        //              startApp();
                        q_startPressed=1;
                        break;
#if 0
                case stopButton:
                        stopApp();
                        break;
                case emptyButton:
                        empty();
                        break;
                case upButton:
                        KeyUp();
                        break;
                case downButton:
```

```
                        KeyDown();
                        break;
                case selectButton:
                        selectItem();
                        break;
#endif
        }

    }
}

void setCountElem(void){

        if (opData[0]!='\0') {
                count_elem = strlen(opData);
#if VERIFICATION
        assert(count_elem>=0);
#endif
        }
}

uData8 getSenPos(void){

    uData8 result=0;

    ++count_pos;
    switch(opData[count_pos-1]){
        case HR:
            result=HR;
            break;
        case SPO2:
            result=SPO2;
            break;
                case EHR:
            result=EHR;
            break;
                case HRD:
            result=HRD;
            break;
                case EHRD:
            result=EHRD;
            break;
                case SPO2D:
            result=SPO2D;
            break;
                case SPO2FAST:
            result=SPO2FAST;
            break;
                case SPO2B:
            result=SPO2B;
            break;
                case ESPO2:
            result=ESPO2;
            break;
                case ESPO2D:
            result=ESPO2D;
            break;
    }

    if (count_pos == count_elem) {
        count_pos = 0;
    }

#if VERIFICATION
        assert(result>=0);
        assert(count_pos>0);
#endif

    return result;

}

uData8 getButtonState(void){

    return enable_buttons;

}
```

LISTING A.4: Source code for Pulse Oximeter file pulse/apps/menu_app.c

```
/**********************************************************
* File:        menu_app.h
* Abstract:    Interface of the application menu
* Platform:    AT89S8252
* Project:     Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
**********************************************************/

/*******************
* INCLUDE FILES    *
********************/
#if (TARGET)
#include <REG51.H>
#endif

#include "../drivers/global.h"
#include "../drivers/sensor.h"
#include "../drivers/lcd_driver.h"
#include "../drivers/keyboard.h"
#include "../utils/log.h"

/*******************
* EXPORTED MACROS  *
********************/
/**
 * @brief indicate the amount of sensor data to be displayed.
 */
#define AMOUNTOFDATA    (11)

/***********************************************
* ENUMERATIONS                                                *
************************************************/
enum MenuSates { SETSAMPLETIME=1, SETLOG, TRANSFERLOG, SETHR,
                                SETSPO2, CONCABLE, SETHRD, SETEHRD, SETEHR,
                                SETSPO2D, SETSPO2FAST, SETSPO2B, SETESPO2, SETESPO2D,
                                SETSREV };

/*******************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
/**
 *   @brief Function used to initialize the menu internal variables.
 *
 *   @retval void
 */
extern void initMenuApp(void);

/**
 *   @brief Function used to get the sensor data that must be shown
 *                     currently to the user.
 *
 *   @retval The sensor data to be shown. Otherwise, -1 is returned.
 */
extern uData8 getSenPos(void);

/**
 *   @brief Function used to know which buttons are enables at a given moment.
 *
 *   @retval uData8
 */
extern uData8 getButtonState(void);

/**
 *   @brief Function used to choose which sensor data will be displayed to the user.
 *
 *   @retval 0 success, else failure.
 */
extern Data8 chooseSensorData(uData8 op, uData8 en);

/**
 *   @brief Function used to set the amount of elements to be displayed on the display.
 *
 *   @retval void
 */
extern void setCountElem(void);

/**
 *   @brief Function used to calculate the value of the progress unit.
```

```
 *
 *   @retval The value of the progress unit. Otherwise, -1 is returned.
 */
extern uData8 calculateUnitProgress(void);

/**
 *   @brief Function used to select the data item that will be shown to the user.
 *
 *   @retval The value of the menu state. Otherwise, -1 is returned.
 */
extern uData8 selectItem(void);

/**
 *   @brief Function used to increase the value of a menu item.
 *
 *   @retval The value of the item that has been increased. Otherwise, -1 is returned.
 */
extern uData8 KeyUp(void);

/**
 *   @brief Function used to deacrease the value of a menu item.
 *
 *   @retval The value of the item that has been deacreased. Otherwise, -1 is returned.
 */
extern uData8 KeyDown(void);
```

LISTING A.5: Source code for Pulse Oximeter file pulse/apps/menu_app.h

```
/*******************************************************
* File:        global.h
* Abstract:    Global variables of the system
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/

#ifndef _GLOBAL_H
#define _GLOBAL_H

/*******************
* INCLUDE FILES    *
********************/
#include <string.h>      /* functions to manipulate the strings */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define TRUE  (1)
#define FALSE (0)
#define NULL_POINTER    (0)

/**
*  @brief 0 -> indicates that the software must be
*                 compiled using GCC. 1 -> indicates that
*                 the software must be compiled using sdcc.
*/
#define TARGET (0)

/**
*  @brief 1 -> indicates that the hardware dependent
*                 software will be verified by model checker
*                 tools. 0 -> otherwise.
*/
#define VERIFICATION (1)

#define BIT0 (0x01)
#define BIT1 (0x02)
#define BIT2 (0x04)
#define BIT3 (0x08)
#define BIT4 (0x10)
#define BIT5 (0x20)
#define BIT6 (0x40)
#define BIT7 (0x80)
#define BIT8 (BIT7<<1)

#define LCDSIZE         (32)
#define LINE1 1
#define LINE2 2
```

```c

/********************************
* ENUMERATIONS
********************************/
enum sensorOp { HR=1, SPO2, SPO2D, SPO2FAST, SPO2B, EHR,
                ESPO2, HRD, EHRD, ESPO2D, SREV };


/*****************************************************************
* TYPEDEF DECLARATION                                                              *
*****************************************************************/
typedef unsigned char _uchar;
typedef _Bool _array_of_2_Bool[2];
typedef _Bool _array_of_3_Bool[3];
typedef _Bool _array_of_4_Bool[4];
typedef _Bool _array_of_6_Bool[6];
typedef _Bool _array_of_12_Bool[12];
typedef _Bool _array_of_11_Bool[11];
typedef _array_of_3_Bool _array_of_2__array_of_3_Bool[2];
typedef _array_of_2_Bool _array_of_2__array_of_2_Bool[2];


/*********************
* EXPORTED STRUCT    *
*********************/

struct module_oc8051_uart {
  _Bool rst;
  _Bool clk;
  _Bool bit_in;
  _Bool wr;
  _Bool rxd;
  _Bool wr_bit;
  _Bool t1_ow;
  _Bool brate2;
  _Bool pres_ow;
  _Bool rclk;
  _Bool tclk;
  _uchar data_in;
  _uchar wr_addr;
  _Bool txd;
  _Bool intr;
  _uchar scon;
  _uchar pcon;
  _uchar sbuf;
  _Bool t1_ow_buf;
  _Bool trans;
  _Bool receive;
  _Bool tx_done;
  _Bool rx_done;
  _Bool rxd_r;
  _Bool shift_tr;
  _Bool shift_re;
  _array_of_2_Bool rx_sam;
  _array_of_4_Bool tr_count;
  _array_of_4_Bool re_count;
  _uchar sbuf_rxd;
  _array_of_12_Bool sbuf_rxd_tmp;
  _array_of_11_Bool sbuf_txd;
  _Bool ren;
  _Bool tb8;
  _Bool rb8;
  _Bool ri;
  _Bool smod;
  _Bool wr_sbuf;
  _Bool sc_clk_tr;
  _Bool smod_clk_tr;
  _Bool sc_clk_re;
  _Bool smod_clk_re;
};

struct module_oc8051_tc {
  _uchar wr_addr;
  _uchar data_in;
  _Bool clk;
  _Bool rst;
  _Bool wr;
  _Bool wr_bit;
  _Bool ie0;
  _Bool ie1;
```

```
  _Bool tr0;
  _Bool tr1;
  _Bool t0;
  _Bool t1;
  _Bool pres_ow;
  _uchar tmod;
  _uchar tl0;
  _uchar th0;
  _uchar tl1;
  _uchar th1;
  _Bool tf0;
  _Bool tf1;
  _Bool tf1_0;
  _Bool tf1_1;
  _Bool t0_buff;
  _Bool t1_buff;
  _Bool tc0_add;
  _Bool tc1_add;
};

struct module_oc8051_int {
  _uchar wr_addr;
  _uchar data_in;
  _Bool wr;
  _Bool tf0;
  _Bool tf1;
  _Bool t2_int;
  _Bool ie0;
  _Bool ie1;
  _Bool clk;
  _Bool rst;
  _Bool reti;
  _Bool wr_bit;
  _Bool bit_in;
  _Bool ack;
  _Bool uart_int;
  _Bool tr0;
  _Bool tr1;
  _Bool intr;
  _uchar int_vec;
  _uchar ie;
  _uchar tcon;
  _uchar ip;
  _array_of_4_Bool tcon_s;
  _Bool tcon_tf1;
  _Bool tcon_tf0;
  _Bool tcon_ie1;
  _Bool tcon_ie0;
  _array_of_3_Bool isrc_cur;
  _array_of_2__array_of_3_Bool isrc;
  _array_of_2_Bool int_dept;
  _array_of_2_Bool int_dept_1;
  _Bool int_proc;
  _array_of_2__array_of_2_Bool int_lev;
  _Bool cur_lev;
  _array_of_6_Bool int_l0;
  _array_of_6_Bool int_l1;
  _array_of_6_Bool ip_l0;
  _array_of_6_Bool ip_l1;
  _array_of_6_Bool int_src;
  _Bool il0;
  _Bool il1;
  _Bool tf0_buff;
  _Bool tf1_buff;
  _Bool ie0_buff;
  _Bool ie1_buff;
};


/*******************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/

#endif /* _GLOBAL_H */
```

LISTING A.6: Source code for Pulse Oximeter file pulse/drivers/global.h

```
/*****************************************************************
* File:        keyboard.c
```

```
* Abstract:    Implementation of the keyboard functions
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*****************************************************************/

/*****************************************************************
* INCLUDE FILES                                                                                                 *
*****************************************************************/
#include "keyboard.h"

/*****************************************************************
* ENUMARATIONS                                                  *
*****************************************************************/
enum Key_State {START=BIT0, STOP=BIT1, EMPTY1=BIT2, EMPTY2=BIT3,
                UP=BIT4, DOWN=BIT5, SELECT=BIT6, EMPTY3=BIT7};

/******************************************************************************
* FUNCTION IMPLEMENTATION                                                                                        *
******************************************************************************/

Data8 checkPressedButton(uData8 Key){

    int command=0;

    switch(Key){

                case START:            command=startButton;    break;
                case STOP:             command=stopButton;     break;
                case UP:       command=upButton;       break;
                case DOWN:     command=downButton;     break;
                case SELECT:   command=selectButton;   break;
                case EMPTY1:   command=emptyButton;    break;
                case EMPTY2:   command=emptyButton;    break;
                case EMPTY3:   command=emptyButton;    break;

    }

#if VERIFICATION
        assert(command>=-1);
#endif

    return command;

}
```

LISTING A.7: Source code for Pulse Oximeter file pulse/drivers/keyboard.c

```
/********************************************************
* File:        serial.h
* Abstract:    Interface of the keyboard driver
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
********************************************************/
#ifndef _KEYBOARD_H
#define _KEYBOARD_H

/*******************
* INCLUDE FILES    *
********************/
#include "global.h"
#include "sensor.h"
#include "lcd_driver.h"

/*******************
* ENUMARATIONS     *
*******************/
/**
 * @brief indicate the all available buttons of the keyboard.
 */
enum Key_Value {startButton=1, stopButton, emptyButton, upButton,
                downButton, selectButton};

/*******************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
```

```
/**
*   @brief Function used to detect the key that the user pressed
*
*   @retval Data8
*/
extern Data8 checkPressedButton(uData8);

#endif /* _KEYBOARD_H */
```

LISTING A.8: Source code for Pulse Oximeter file pulse/drivers/keyboard.h

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File:        LCD.a51
; Abstract:    Functions to access the LCD in Assembly
; Platform:    AT89S8252
; Project      Pulse Oximeter
; Author(s):   Lucas Cordeiro
; Copyright (C)2007 DCC, Federal University of Amazonas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.globl _lcd_init

_lcd_init:

;TIMER          EQU 45535                       ;50.000us

                lcall inic_display
                mov a,#0x01
                lcall wrcar
                mov a,#0x80
                lcall wrcar
                mov R1,#0x33


inic_display:

                lcall t5ms
                MOV A,#0x38              ;00111000b   | 0001 1100
                LCALL wrcar             ;                 1  c
                MOV A,#0x06              ; 00000110 | 0110 0000
                LCALL wrcar             ;              6   0
                MOV A,#0x0E              ; 00001110  | 0111 0000
                LCALL wrcar             ;                7   0
                MOV A,#0x01              ; 00000001  | 1000 0000 (Limpa display
                LCALL wrcar             ;               8  0
                MOV A,#0x0C              ;desliga cursor
                LCALL wrcar
                LCALL t5ms
                RET

wrcar:
                push  DPH
                push  DPL

                mov DPTR,#0x8000
                movx  @DPTR,A
                lcall t5ms

                pop   DPL
                pop   DPH
                ret

t5ms:
                MOV TL0,#0xdf
                MOV TH0,#0xb1
                SETB TR0
;                JNB TF0,$
                CLR TR0
                CLR TF0
                RET

                        ; SOURCE LINE # 3
;
; }
                        ; SOURCE LINE # 5
        RET
; END OF lcd_init
```

LISTING A.9: Source code for Pulse Oximeter file pulse/drivers/lcd.asm

```c
/*******************************************************
* File:        LCD.c
* Abstract:    Implementation of the LCD
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/

#include <string.h>     /* functions to manipulate strings */

/**********************************************
* INCLUDE FILES                               *
**********************************************/
#include "lcd_driver.h"

/* Assembly routines called from C program */
extern void write2lcd(unsigned char dado);
extern void lcd_init(void);

/***************************************************************
* STATIC DATA                                                                                         *
***************************************************************/
static int length=0;

/**************************************************************************
* FUNCTION IMPLEMENTATION                                                                                              *
**************************************************************************/
void lcd_write(const char *sPtr, int line, int column){

        size_t size;
        int i, flag_l1=FALSE, flag_l2=FALSE;
        char line1[LCDSIZE/2], line2[LCDSIZE/2];
        const char msg[LCDSIZE/2]="Message too long";

        if (sPtr!=NULL)
                size = strlen(sPtr);

        for(i=0; i<LCDSIZE/2; i++){
                line1[i]='\0';
                line2[i]='\0';
        }

        if(size<=LCDSIZE/2){
                flag_l1=1;
                for(i=0; i<LCDSIZE/2; i++){
                        line1[i]=sPtr[i];
                }
        }
        else if (size>LCDSIZE/2 && size<=LCDSIZE){
                flag_l1=1;
                flag_l2=1;
                for(i=0; i<size; i++){

                        if(i<LCDSIZE/2){
                                line1[i]=sPtr[i];
                        }
                        else{
                                line2[i-(LCDSIZE/2)]=sPtr[i];
                        }
                }

        }
        else{
                flag_l1=1;
                for(i=0; i<(LCDSIZE/2); i++){
                        line1[i]=msg[i];
                }
        }

        if(flag_l1==TRUE){
                lcd_printf(line1, line, column);
        }
        if(flag_l2==TRUE){
                lcd_printf(line2, line, column);
        }

}
```

```
/**
 * @brief Clean the display
 *
 * @retval void
 *
 * Comments:
 * - This procedures initializes the LCD in order to allow
 *   the programmer to write new strings on the LCD.
 */

uData8 lcd_clean(void){

#if (TARGET)
        lcd_init();
#endif

        length=0;

        return LCD_OK;
}

/**
 * @brief: Write text to LCD
 *
 * retval: void
 *
 * Comments:
 * - This routine outputs some text to the LCD display
 *   according to the line and column parameters passed
 *   by the programmer.
 */

uData8 lcd_printf(const char *sPtr, _Bool line, int column)
{

        int i,j;

        if (line==1){
           for(i=0; i<(column-1); i++){
#if (TARGET)
              write2lcd(0x20); /* write "null" to LCD */
#endif
           }
        }
        else if (line == 2){
           for(i=0; i<(40-length); i++){
#if (TARGET)
              write2lcd(0x20); /* write "null" to LCD */
#endif
           }
           for(j=0; j<(column-1); j++){
#if (TARGET)
              write2lcd(0x20); /* write "null" to LCD */
#endif
           }
        }

        if (sPtr != NULL) {
            for ( ; *sPtr != '\0'; sPtr++ ){
        #if (TARGET)
                    write2lcd(*sPtr);
        #endif
                        length++;
           }
        } else {
                return NULL_POINTER;
        }

        return length;
}
```

LISTING A.10: Source code for Pulse Oximeter file pulse/drivers/lcd_driver.c

```
/*******************************************************
* File:        LCD.h
* Abstract:    Interface of the LCD
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
```

```
* Copyright (C)2007 DCC, Federal University of Amazonas
********************************************************/

#ifndef _LCD_DRIVER_H
#define _LCD_DRIVER_H

/*******************
* INCLUDE FILES    *
*******************/
#include "global.h"
#include "sensor.h"

#define LCD_OK  (1)

/********************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
/**
*   @brief Function used to write text to LCD 16x2
*
*   @retval void
*/
uData8 lcd_printf(const char *sPtr, _Bool line, int column);
//inconsistency found by NuSMV2
//void lcd_printf(const char *sPtr, int line, int column);

/**
*   @brief Function used to clean the LCD 16x2
*
*   @retval void
*/
extern uData8 lcd_clean(void);

#endif /* _LCD_DRIVER_H */
```

LISTING A.11: Source code for Pulse Oximeter file pulse/drivers/lcd_driver.h

```
/*****************************************************************
* File:        sensor.c
* Abstract:    Implementation of the sensor interface
* Platform:    AT89S8252
* Project:     Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
******************************************************************/

/**********************************************
* INCLUDE FILES                               *
**********************************************/
#include "sensor.h"
#include "../utils/log.h"

/*****************************************************************
* STATIC FUNCTION PROTOTYPES                                     *
******************************************************************/
static void fillArrays(Data8 rawData, uData8 cont);

/*****************************************************************
* STATIC DATA                                                    *
******************************************************************/
static Data8 itr=0, checkSum[SIZEOFFRAME];
static uData8 contPos=0, contCHK=0, flag1=FALSE, flag2=FALSE, frame=0;
static char serialData[MAXNUMOFBYTE];
static char tmp[10];
static uData8 srev;     /* Oximeter Firmware Revision Level */
#if (TARGET)
bit OutofTrack=FALSE;
#else
static Data8 OutofTrack=FALSE;
#endif

/****************************************************************************
* FUNCTION IMPLEMENTATION                                                   *
*****************************************************************************/
void initSensor(void){

        Data8 i;

        df2_ptr = &df2;
```

```
        sf_ptr = &sf;
        avg_ptr = &avg;

        for(i=0; i<ELEMEN; i++){

                df2_ptr->hrmsb[i]=0;
                df2_ptr->hrlsb[i]=0;
                df2_ptr->spo2[i]=0;
                df2_ptr->spo2d[i]=0;
                df2_ptr->spo2fast[i]=0;
                df2_ptr->spo2b[i]=0;
                df2_ptr->ehrmsb[i]=0;
                df2_ptr->ehrlsb[i]=0;
                df2_ptr->espo2[i]=0;
                df2_ptr->hrdmsb[i]=0;
                df2_ptr->hrdlsb[i]=0;
                df2_ptr->ehrdmsb[i]=0;
                df2_ptr->ehrdlsb[i]=0;
                df2_ptr->espo2d[i]=0;

        }

}

#if 1
void initStatus(void){

        if (sf_ptr != NULL)     {
                sf_ptr->sync = FALSE;
                sf_ptr->gprf = FALSE;
                sf_ptr->rprf = FALSE;
                sf_ptr->yprf = FALSE;
                sf_ptr->snsa = FALSE;
                sf_ptr->oot  = FALSE;
                sf_ptr->artf = FALSE;
                sf_ptr->snsd = FALSE;
                sf_ptr->bit7 = FALSE;
        }
}
#else
//bug
void initStatus(void){

        sf_ptr->sync = FALSE;
        sf_ptr->gprf = FALSE;
        sf_ptr->rprf = FALSE;
        sf_ptr->yprf = FALSE;
        sf_ptr->snsa = FALSE;
        sf_ptr->oot  = FALSE;
        sf_ptr->artf = FALSE;
        sf_ptr->snsd = FALSE;
        sf_ptr->bit7 = FALSE;

}
#endif

/**
 * @biref: Calculate the average of the sensor data
 * @retval: integer
 *
 * Comments:
 * - This function returns the average value of the data
 *   read by the sensor. For instance, the sensor provides
 *   the HR and SpO2 three times within one second then
 *   this function sums three HR or SpO2 values and
 *   divides them by the amount was read.
 */
uData8 showAverage(Data8 *sensorData){

        Data8 i=0, sensorValue=0, numElements=0, aux=0;
#if VERIFICATION
        __CPROVER_assume(sensorData!=NULL);
#endif
        for(i=0; i<ELEMEN; i++){
                if ( sensorData[i]!=0 ){
                        sensorValue = sensorValue + sensorData[i];
                        ++numElements;
                }
        }
        if (numElements!=0){
```

```
                aux = sensorValue/numElements;
        }
        else{
                aux=0;
        }

#if VERIFICATION
        assert(aux>=0);
#endif

        return aux;

}

uData8 signalInverter(Data8 signal) {

        uData8 inverter;

        if(signal >= 0) {
                inverter = signal;
        } else {
                inverter = INV*signal;
        }

#if VERIFICATION
        assert(inverter>=0);
#endif

        return inverter;
}

uData8 checkValidBytes(Data8 *chBytes) {

        uData8 i, chSum=0, result=0, err=0;

        for(i=0; i<(SIZEOFFRAME-1); i++) {
                chSum = chSum + signalInverter(chBytes[i]);
        }

        result = (chSum%BIT8);

        if(result!=chBytes[SIZEOFFRAME-1]) {
                err=-1;
        }

        return err;
}

/**
 * @brief: Collect the data from the sensor
 * Return value: none
 *
 * Comments:
 * - This procedures is called when there is data available
 *   in the serial port. Then it stores the data from the sensor
 *   in an array that will be used further in order to fill in
 *   other arrays.
 */
void collectData(Data8 sensorByte) {

        int chkerr, i;

        debug("%dSENSOR->sensor.c:%d(%d), Testing...", TOKEN, __FUNCTION__, __LINE__);

        if ( contPos==125) {
            contPos=0;
                frame=0;
                flag1=FALSE;
                flag2=FALSE;
        }

        if (sensorByte == 1 || flag2 == TRUE) {

                if (flag2 == FALSE) {
                        checkSum[contCHK]=sensorByte;
                        contCHK++;
                        flag2=TRUE;
                }
                else if ((SYNC&sensorByte) == TRUE || flag1 == TRUE) {
                        checkSum[contCHK]=sensorByte;
```

```
                        contCHK++;
                }       flag1=TRUE;
        }
        if(frame==3){
                srev = checkSum[3];
        }

        if (contCHK == 5){
                chkerr = checkValidBytes(checkSum);
                if (chkerr == FALSE){
                        for(i=0; i<SIZEOFFRAME; i++){
                                fillArrays(checkSum[i], contPos);
                                contPos++;
                        }
                }
                else{
                        for(i=0; i<SIZEOFFRAME; i++){
                                if(i==3){
                                        fillArrays(0, contPos);
                                        contPos++;
                                }
                                else{
                                        fillArrays(checkSum[i], contPos);
                                        contPos++;
                                }
                        }
                }
                frame++;
                contCHK=0;
        }


#if VERIFICATION
        assert(contPos>=0);
        assert(contCHK>=0);
#endif

}

#if 1
uData8 sensorCheckStatus(Data8 statusByte){

        if (sf_ptr != NULL)
        {
                sf_ptr->sync = SYNC&statusByte;
                sf_ptr->gprf = GPRF&statusByte;
                sf_ptr->rprf = RPRF&statusByte;
                sf_ptr->yprf = YPRF&statusByte;
                sf_ptr->snsa = SNSA&statusByte;
                sf_ptr->oot  = OOT&statusByte;
                sf_ptr->artf = ARTF&statusByte;
                sf_ptr->snsd = SNSD&statusByte;
                sf_ptr->bit7 = SBIT7&statusByte;
        } else {
                return NULL_POINTER;
        }

        return SENSOR_OK;
}
#else
//bug
static void sensorCheckStatus(Data8 statusByte){

        sf_ptr->sync = SYNC&statusByte;
        sf_ptr->gprf = GPRF&statusByte;
        sf_ptr->rprf = RPRF&statusByte;
        sf_ptr->yprf = YPRF&statusByte;
        sf_ptr->snsa = SNSA&statusByte;
        sf_ptr->oot  = OOT&statusByte;
        sf_ptr->artf = ARTF&statusByte;
        sf_ptr->snsd = SNSD&statusByte;
        sf_ptr->bit7 = SBIT7&statusByte;

}
#endif

/**
 * @brief: Fill in arrays with sensor data
 * Return value: none
 *
```

```
 * Comments:
 * - This procedures is called in order to fill in
 *   the heart rate and SpO2 arrays in standard and
 *   display modes. When SpO2 and HR cannot be computed,
 *   the sensor will send a missing data indicator.
 *   For missing data, the HR equals 511 and the
 *   SpO2 equals 127.
 */

static void fillArrays(Data8 rawData, uData8 cont){

        if (cont==124){           /* verify if the packet is complete with 25 frames */
                itr++;
        }

        if (itr==3){             /* check if three packets were already read */
        itr=0;
            setSensorData();
            //printValue();
     }

        if (df2_ptr != NULL)
        {
                switch(cont){

                        case posStatus:
                                sensorCheckStatus(rawData);
                                break;
                        case posHRMSB:
                                if (rawData != 3){
                                        df2_ptr->hrmsb[itr]= rawData*BIT7;
                                }
                                else{
                                        df2_ptr->hrmsb[itr]=0;
                                }
                                break;
                        case posHRLSB:
                                if (rawData != 127){
                                        df2_ptr->hrlsb[itr]= rawData;
                                }
                                else{
                                        df2_ptr->hrlsb[itr]=0;
                                }
                                break;
                        case posSpO2:
                                if (rawData != 127){
                                        df2_ptr->spo2[itr]= rawData;
                                }
                                else{
                                        df2_ptr->spo2[itr]=0;
                                }
                                break;
#if 0
                case posREV:
                        srev = rawData;
                        break;
#endif
                        case posSpO2D:
                                if (rawData != 127){
                                        df2_ptr->spo2d[itr]= rawData;
                                }
                                else{
                                        df2_ptr->spo2d[itr]=0;
                                }
                                break;
                        case posSpO2Fast:
                                if (rawData != 127){
                                        df2_ptr->spo2fast[itr]= rawData;
                                }
                                else{
                                        df2_ptr->spo2fast[itr]=0;
                                }
                                break;
                        case posSpO2BB:
                                if (rawData != 127){
                                        df2_ptr->spo2b[itr]= rawData;
                                }
                                else{
                                        df2_ptr->spo2b[itr]=0;
                                }
```

```
                                        break;
                                case posEHRMSB:
                                        if (rawData != 3){
                                                df2_ptr->ehrmsb[itr]= rawData*BIT7;
                                        }
                                        else{
                                                df2_ptr->ehrmsb[itr]=0;
                                        }
                                        break;
                                case posEHRLSB:
                                        if (rawData != 127){
                                                df2_ptr->ehrlsb[itr]= rawData;
                                        }
                                        else{
                                                df2_ptr->ehrlsb[itr]=0;
                                        }
                                        break;
                                case posESpO2:
                                        if (rawData != 127){
                                                df2_ptr->espo2[itr]= rawData;
                                        }
                                        else{
                                                df2_ptr->espo2[itr]=0;
                                        }
                                        break;
                                case posESpO2D:
                                        if (rawData != 127){
                                                df2_ptr->espo2d[itr]= rawData;
                                        }
                                        else{
                                                df2_ptr->espo2d[itr]=0;
                                        }
                                        break;
                                case posHRDMSB:
                                        if (rawData != 3){
                                                df2_ptr->hrdmsb[itr]= rawData*BIT7;
                                        }
                                        else{
                                                df2_ptr->hrdmsb[itr]=0;
                                        }
                                        break;
                                case posHRDLSB:
                                        if (rawData != 127){
                                                df2_ptr->hrdlsb[itr]= rawData;
                                        }
                                        else{
                                                df2_ptr->hrdlsb[itr]=0;
                                        }
                                        break;
                                case posEHRDMSB:
                                        if (rawData != 3){
                                                df2_ptr->ehrdmsb[itr]= rawData*BIT7;
                                        }
                                        else{
                                                df2_ptr->ehrdmsb[itr]=0;
                                        }
                                        break;
                                case posEHRDLSB:
                                        if (rawData != 127){
                                                df2_ptr->ehrdlsb[itr]= rawData;
                                        }
                                        else{
                                                df2_ptr->ehrdlsb[itr]=0;
                                        }
                                        break;
                        }
                }
#if VERIFICATION
        assert(itr<4);
#endif
}

/**
 * @brief This sets all HR and SpO2 data in standard
 *        and display mode.
 *
 * Comments: This procedure is called every second.
 */
uData8 setSensorData(void){

```

```
	if (df2_ptr != NULL)
	{
		avg_ptr->hr = showAverage(df2_ptr->hrmsb) + showAverage(df2_ptr->hrlsb);
		avg_ptr->hrd = (showAverage(df2_ptr->hrdmsb) + showAverage(df2_ptr->hrdlsb));
		avg_ptr->ehrd = (showAverage(df2_ptr->ehrdmsb) + showAverage(df2_ptr->ehrdlsb));
		avg_ptr->ehr = (showAverage(df2_ptr->ehrmsb) + showAverage(df2_ptr->ehrlsb));
		avg_ptr->spo2 = showAverage(df2_ptr->spo2);
		avg_ptr->spo2d = showAverage(df2_ptr->spo2d);
		avg_ptr->spo2fast = showAverage(df2_ptr->spo2fast);
		avg_ptr->spo2b = showAverage(df2_ptr->spo2b);
		avg_ptr->espo2 = showAverage(df2_ptr->espo2);
		avg_ptr->espo2d = showAverage(df2_ptr->espo2d);
	} else {
		return NULL_POINTER;
	}

	return SENSOR_OK;
}

uData8 getHR(void){

	if (avg_ptr !=NULL)
		return avg_ptr->hr;	/* Provide the HR value in standard mode */
	else
		return NULL_POINTER;
}

uData8 getHRD(void){

	if (avg_ptr !=NULL)
		return avg_ptr->hrd;	/* Provide the HR value in display mode */
	else
		return NULL_POINTER;

}

uData8 getEHRD(void){

	if (avg_ptr !=NULL)
		return avg_ptr->ehrd;	/* Provide the EHRD value in display mode */
	else
		return NULL_POINTER;
}

uData8 getEHR(void){

	if (avg_ptr !=NULL)
		return avg_ptr->ehr;	/* Provide the EHR value in standard mode */
	else
		return NULL_POINTER;
}

uData8 getSpO2(void){

	if (avg_ptr !=NULL)
		return avg_ptr->spo2;	/* Provide the SpO2 value in standard mode */
	else
		return NULL_POINTER;
}

uData8 getSpO2D(void){

	if (avg_ptr !=NULL)
		return avg_ptr->spo2d;	/* Provide the SpO2 value in display mode */
	else
		return NULL_POINTER;
}

uData8 getSpO2Fast(void){

	if (avg_ptr !=NULL)
		return avg_ptr->spo2fast;	/* Provide the SpO2 Fast value in standard mode */
	else
		return NULL_POINTER;
}

uData8 getSpO2B(void){

	if (avg_ptr !=NULL)
		return avg_ptr->spo2b;	/* Provide the SpO2B-B value in standard mode */
```

```
        else
                return NULL_POINTER;
}

uData8 getESpO2(void){

        if (avg_ptr !=NULL)
                return avg_ptr->espo2; /* Provide the ESpO2 value in standard mode */
        else
                return NULL_POINTER;
}

uData8 getESpO2D(void){

        if (avg_ptr!=NULL)
                return avg_ptr->espo2d; /* Provide the ESpO2D value in standard mode */
        else
                return NULL_POINTER;
}

uData8 getSREV(void){

        return srev;    /* Provide the firmware version */

}


#if (TARGET)
bit IsOutofTrack(){

   return OutofTrack;   /* indicates if the sensor is out of track or not */

}

bit IsSensorDisconnected(void){

    return sf_ptr->snsd;

}

#else

uData8 IsOutofTrack(){

   return OutofTrack;   /* indicates if the sensor is out of track or not */

}

uData8 IsSensorDisconnected(void){

        if (sf_ptr != NULL)
        return sf_ptr->snsd;
        else
                return NULL_POINTER;

}

#endif
```

LISTING A.12: Source code for Pulse Oximeter file pulse/drivers/sensor.c

```
/*******************************************************
* File:        sensor.h
* Abstract:    Interface of the sensor driver
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/
#ifndef _SENSOR_H
#define _SENSOR_H

/*******************
* INCLUDE FILES    *
********************/
#include "global.h"

/*******************
* EXPORTED MACROS  *
```

```
********************/
#define ELEMEN 3  /* define the number of elements in the sensor data array */
#define MAXNUMOFBYTE    (125)
#define VALIDDATA               (1)
#define SIZEOFFRAME             (5)
#define INV                             (-1)
#define SENSOR_OK               (1)

/*********************
* EXPORTED TYPEDEFS  *
**********************/
typedef int Data8;
typedef unsigned int uData8;

/*********************
* ENUMERATIONS       *
**********************/
/* This enum defines the position of the sensor data in the packets */
enum sensorPosition {posStatus=1, posPleth=2, posHRMSB=3, posCHKSUM=4, posHRLSB=8,
                                  posSpO2=13, posREV=18, posSpO2D=43, posSpO2Fast=48,
                                  posSpO2BB=53, posEHRMSB=68, posEHRLSB=73, posESpO2=78,
                                  posESpO2D=83, posHRDMSB=98, posHRDLSB=103,
                                  posEHRDMSB=108, posEHRDLSB=113};

enum sensorStatus {SYNC=0x01, GPRF=0x02, RPRF=0x03, YPRF=0x07, SNSA=0x08,
                        OOT=0x10, ARTF=0x20, SNSD=0x40, SBIT7=0x80};


/*****************************************************************
* EXTERNAL STRUCTS                                                *
******************************************************************/
/* Standard: SpO2 and HR updated on every pulse beat. */
/* Display: SpO2 and HR updated every 1.5 seconds. */

struct dataFormat{

        Data8 hrmsb[ELEMEN];    /* 4-beat average values in standard mode (MSB). */
        Data8 hrlsb[ELEMEN];    /* 4-beat average values in standard mode (LSB). */
        Data8 spo2[ELEMEN];             /* 4-beat average values in standard mode.       */
        Data8 spo2d[ELEMEN];    /* 4-beat average displayed values in display mode */
        Data8 spo2fast[ELEMEN]; /* Non-slew limited saturation with 4-beat averaging in standard mode. */
        Data8 spo2b[ELEMEN];    /* Un-averaged, non-slew limited, beat to beat value in standard mode. */
        Data8 ehrmsb[ELEMEN];   /* 8-beat average values in standard mode. */
        Data8 ehrlsb[ELEMEN];   /* 8-beat average values in standard mode. */
        Data8 espo2[ELEMEN];    /* 8-beat average values in standard mode. */
        Data8 hrdmsb[ELEMEN];   /* 4-beat average displayed values in display mode (MSB) */
        Data8 hrdlsb[ELEMEN];   /* 4-beat average displayed values in display mode (LSB) */
        Data8 ehrdmsb[ELEMEN];  /* 8-beat average displayed values in display mode (MSB) */
        Data8 ehrdlsb[ELEMEN];  /* 8-beat average displayed values in display mode (LSB) */
        Data8 espo2d[ELEMEN];   /* 8-beat average displayed values in display mode */

} df2;


struct sensorAvgVal{

        Data8 hr;
        Data8 spo2;
        Data8 spo2d;
        Data8 spo2fast;
        Data8 spo2b;
        Data8 ehr;
        Data8 espo2;
        Data8 hrd;
        Data8 ehrd;
        Data8 espo2d;

} avg;

struct statusFormat{

#if (TARGET)
        bit sync;
        bit gprf;
        bit rprf;
        bit yprf;
        bit snsa;
        bit oot;
        bit artf;
        bit snsd;
```

```
        bit bit7;
#else
        uData8 sync;
        uData8 gprf;
        uData8 rprf;
        uData8 yprf;
        uData8 snsa;
        uData8 oot;
        uData8 artf;
        uData8 snsd;
        uData8 bit7;
#endif
} sf;

struct dataFormat *df2_ptr;
struct statusFormat *sf_ptr;
struct sensorAvgVal *avg_ptr;

/*******************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
extern uData8 showAverage(Data8 sensorData[]);
extern uData8 getHR(void);
extern uData8 getHRD(void);
extern uData8 getEHRD(void);
extern uData8 getEHR(void);
extern uData8 getSpO2(void);
extern uData8 getSpO2D(void);
extern uData8 getSpO2Fast(void);
extern uData8 getSpO2B(void);
extern uData8 getESpO2(void);
extern uData8 getESpO2D(void);
extern uData8 getSREV(void);
extern uData8 checkValidBytes(Data8 *chBytes);
extern uData8 signalInverter(Data8 signal);
extern void initSensor(void);
extern uData8 setSensorData(void);
extern uData8 checkStatus(void);

#if (TARGET)
   bit IsOutofTrack(void);
   bit IsSensorDisconnected(void);
   bit IsSensorAlarmOn(void);
#else
   extern uData8 IsOutofTrack(void);
   extern uData8 IsSensorDisconnected(void);
   extern uData8 IsSensorAlarmOn(void);
#endif

extern void collectData(Data8 sensorData);
extern void initStatus(void);
extern void printValue(void);

#endif /* _SENSOR_H */
```

LISTING A.13: Source code for Pulse Oximeter file pulse/drivers/sensor.h

```
/*******************************************************
* File:        serial.c
* Abstract:    Implementation of the serial driver
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
********************************************************/

/**********************************************
* INCLUDE FILES                               *
***********************************************/
#include "serial.h"

#define OC8051_RST_SCON 0x00    // serial control
#define OC8051_SFR_SCON 0x98    // serial control 0
#define OC8051_SFR_B_SCON 0x13  // serial control

/*****************************************************************
* FUNCTION PROTOTYPES                                                              *
******************************************************************/
static void Comm(void);
```

```
void next_timeframe(); /* Next Timeframe  */

/*****************************************************************
* EXTERN DATA                                                               *
******************************************************************/
extern const unsigned int bound; /* Unwinding Bound */

/*****************************************************************
* STATIC DATA                                                               *
******************************************************************/
static char sensorword; /* Array with receicved bytes */
static uData8 cont=0;   /* Count received bytes       */

/*****************************************************************
* EXTERN STRUCTS                                                *
******************************************************************/

extern const struct module_oc8051_uart oc8051_uart;
extern const struct module_oc8051_tc oc8051_tc;
extern const struct module_oc8051_int oc8051_int;

/******************************************************************************
* FUNCTION IMPLEMENTATION                                                                    *
*******************************************************************************/
uData8 calculateTimerVal(uData8 BR){

        Data8 timerVal=-1;

    switch(BR){

        case br1200:
            timerVal = reg1200;
            break;
        case br2400:
            timerVal = reg2400;
            break;
        case br9600:
            timerVal = reg9600;
            break;
        case br19200:
            timerVal = reg19200;
            break;
    }

        return timerVal;

}

/**
 * @brief configure the serial port to the
 *   baud rate passed in the function call,
 *   no parity, no flow control, stop bits equal to 1,
 *   and data bits equal to 8.
 */
void serial_init(uData8 baudRate){

        int cycle;
        unsigned char scon_test,wr_addr_bit;

#if (TARGET)

        SCON = 0x50;            /* SCON mode 1, 8-bit UART */
        TMOD = 0x20;            /* TMOD: timer 1, mode 2, 8-bit automatic reload */
        TR1 = 1;                        /* TR1: enable timer 1 */
        IE = 0x90;                      /* enable serial interruption */
        TH1 = calculateTimerVal(baudRate);

#endif

#if (VERIFICATION)

        __CPROVER_assume(oc8051_uart.scon=0x50);
        __CPROVER_assume(oc8051_tc.tmod=0x20);
        __CPROVER_assume(oc8051_int.tr1=1);
        __CPROVER_assume(oc8051_int.ie=0x90);
        __CPROVER_assume(oc8051_tc.th1 =calculateTimerVal(baudRate));
        __CPROVER_assume(oc8051_uart.rst==1);

        wr_addr_bit = oc8051_uart.wr_addr&0x07;

```

```
        for(cycle=0; cycle<bound; cycle++) {
                if (oc8051_uart.rst) {
                scon_test = OC8051_RST_SCON;
                } else if ((oc8051_uart.wr) && !(oc8051_uart.wr_bit) && (oc8051_uart.wr_addr==OC8051_SFR_SCON)) {
                        scon_test = oc8051_uart.data_in;
                } else if ((oc8051_uart.wr) && (oc8051_uart.wr_bit) && (((oc8051_uart.wr_addr&0xF8)>>3)==OC8051_SFR_B_SCON)) {
                        scon_test = oc8051_uart.bit_in;
                } else if (oc8051_uart.tx_done) {
                        scon_test = scon_test|0x02;
                } else if (!oc8051_uart.rx_done) {
                        if ((oc8051_uart.scon&0xC0) == 0x00) {
                                scon_test = scon_test|0x01;
                        } else if ((oc8051_uart.sbuf_rxd_tmp[11]) || !(oc8051_uart.scon&0x20)) {
                                scon_test = scon_test|0x01;
                        scon_test = (scon_test|((scon_test&0x40)&oc8051_uart.sbuf_rxd_tmp[11]));
                        } else {
                        scon_test = (scon_test|((scon_test&0x40)&oc8051_uart.sbuf_rxd_tmp[11]));
                        }
                }
        next_timeframe();
                assert(oc8051_uart.scon==(scon_test&0xFF));
        }

        assert(oc8051_tc.tmod==0x20);
        assert(oc8051_int.tr1==1);
        assert(oc8051_int.ie==0x90);

#endif


}

/**
 * @brief serial interruption. This is called if there are sensor
 * data available in the serial port.
 */

#if (TARGET)
static void Comm(void) interrupt 4 { /* Routine to handle the serial interruption */
        if (RI){
                RI=0;                                   /* set the received flag */
                sensorword=SBUF;                /* read buffer that contains the sensor data */
                collectData(sensorword);
        }


}
#endif

#if (VERIFICATION)
static void Comm(void) { /* Routine to handle the serial interruption */

        if (oc8051_uart.ri){
                oc8051_uart.ri=0;                                       /* set the received flag */
                sensorword=oc8051_uart.sbuf;            /* read buffer that contains the sensor data */
                collectData(sensorword);
                assert(oc8051_uart.ri==0);
        }
}
#endif
```

LISTING A.14: Source code for Pulse Oximeter file pulse/drivers/serial.c

```
/*******************************************************
* File:         serial.h
* Abstract:     Interface of the serial driver
* Platform:     AT89S8252
* Project       Pulse Oximeter
* Author(s):    Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/

#ifndef _SERIAL_H
#define _SERIAL_H

/*******************
* INCLUDE FILES    *
*******************/
#include "global.h"
```

```
#include "sensor.h"

#if (TARGET)
#include <8051.h>
#endif

/********************************
* ENUMERATIONS                                  *
********************************/
/**
 * @brief indicate the baud rates that can be set.
 */
enum BaudRateT {br1200=1200, br2400=2400, br9600=9600,
                    br19200=19200};

/**
 * @brief provide the register value for each baud rate.
 */
enum BaudRateReg {reg1200=0xE8, reg2400=0xF4,
                    reg9600=0xFD, reg19200=0xFD};


/********************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
/**
*   @brief Function used to initialize the serial communication
*
*   @retval void
*/
extern void serial_init(uData8 baudRate);

/**
*   @brief Function used to calculate the value of the TH01 register
*
*   @retval The register value of the baud rate
*/
extern uData8 calculateTimerVal(uData8 baudRate);
#endif /* _SERIAL_H */
```

LISTING A.15: Source code for Pulse Oximeter file pulse/drivers/serial.h

```
/**********************************************************
* File:         timer.c
* Abstract:     Implementation of the timer
* Platform:     AT89S8252
* Project       Pulse Oximeter
* Author(s):    Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
**********************************************************/

/**********************************************
* INCLUDE FILES                               *
**********************************************/
#include "timer.h"

/**********************************************
* LOCAL MACROS                                *
**********************************************/
#define THIGH           (0x3C)
#define TLOW            (0xAF)
#define MAXCOUNTER      (65535)
#define TC              (1000)
#define MASKMSB         (0xFF00)
#define MASKLSB         (0x00FF)
#define ONESECOND       (20)
#define SECOND          (1000)

/****************************************************************
* VERILOG EXTERN STRUCTS                                        *
****************************************************************/

extern const struct module_oc8051_int oc8051_int;
extern const struct module_oc8051_tc oc8051_tc;


extern const struct module_DW8051_intr_1 DW8051_intr_1;
extern const struct module_DW8051_intr_0 DW8051_intr_0;
extern const struct module_DW8051_timer DW8051_timer;
```

```

/*****************************************************************
* STATIC FUNCTION PROTOTYPES                              *
*****************************************************************/
static void system_tick(void);
void next_timeframe();

/*****************************************************************
* EXTERN DATA                                                             *
*****************************************************************/
extern const unsigned int bound; /* Unwinding Bound */

/*****************************************************************
* STATIC DATA                                                             *
*****************************************************************/
static uData8 count_ticks, ticks, tms;

/******************************************************************************
* FUNCTION IMPLEMENTATION                                                                 *
******************************************************************************/
void initTimers(void) {
        count_ticks=0;
        ticks=0;
        tms=0;
}

uData8 calculateTRegVal(uData8 t) {

        uData8 tms;

        if ((t*TC) < MAXCOUNTER )
                tms = MAXCOUNTER - (t*TC);
        else
                tms = 0;

#if VERIFICATION
        assert(tms>=0);
#endif

        return tms;
}

#if 0
bug
uData8 calculateTRegVal(uData8 t) {

        uData8 tms;

        tms = MAXCOUNTER - (t*TC);

        return tms;
}
#endif

uData8 calculateTH(uData8 t) {

        uData8 tReg, result;

        tReg = calculateTRegVal(t);
        result = ((tReg&MASKMSB) >> 8);

#if VERIFICATION
        assert(result>=0);
#endif

        return result;

}

uData8 calculateTL(uData8 t) {

        uData8 tReg, result;

        tReg = calculateTRegVal(t);
        result = tReg&MASKLSB;

#if VERIFICATION
        assert(result>=0);
#endif

```

```
        return result;
}

/**
 * @brief configure the timer according to the parameter passed
 *   in the function call. The time parameter must be in the range
 *   of 1 ms to 50ms.
 */
Data8 initTimer0ms(uData8 time_ms) {

        int cycle;

        tms=time_ms;

#if (TARGET)
        EA = 0;                         /* disable all interruptions (IE register)      */
        TR0 = 0;                        /* stop timer0 (TCON register)                  */
        TMOD = (TMOD&0xF0)|0x01;        /* timer0 mode 1 (TMOD register)                */
        TH0 = calculateTH(time_ms);     /* load TH0 with high order byte (TH)   */
        TL0 = calculateTL(time_ms);     /* load TL0 with low order byte (TLO)   */
        TR0 = 1;                        /* enable counter of timer0 (TCON register)             */
        ET0 = 1;                        /* enable interruption of timer0 (IE register)  */
        EA = 1;                         /* enable all interruptions (IE register)       */
#endif

#if (VERIFICATION)
        __CPROVER_assume(oc8051_int.ie==(oc8051_int.ie&0x7F));
        __CPROVER_assume(oc8051_int.tr0==0);
        __CPROVER_assume(oc8051_tc.tmod==((oc8051_tc.tmod&0xF0)|0x01));
        __CPROVER_assume(oc8051_tc.th0==calculateTH(time_ms));
        __CPROVER_assume(oc8051_tc.tl0==calculateTL(time_ms));
        __CPROVER_assume(oc8051_tc.tr0==1);
        __CPROVER_assume(oc8051_int.ie==(oc8051_int.ie|0x02));
        __CPROVER_assume(oc8051_int.ie==(oc8051_int.ie|0x80));

        for(cycle=0; cycle<bound; cycle++) {
                __CPROVER_assume(oc8051_tc.rst==0);
                next_timeframe();
        }

        assert(oc8051_tc.th0==calculateTH(time_ms));
        assert(oc8051_tc.tl0==calculateTL(time_ms));
        assert((oc8051_tc.tmod&0x01)==0x01);
        assert((oc8051_int.ie&0x82)==0x82);
        assert(oc8051_tc.tr0==1);
#endif

        return (tms <= 0) ? -1:tms;
}

/**
 * @brief configure the timer in scale of seconds.
 *
 */
Data8 initTimer0s(uData8 time_s) {

        if (tms==0) {
                initTimer0ms(50);
                ticks = (time_s*ONESECOND);
        } else if (tms>0){
                ticks = (time_s*(SECOND/tms));
        }

        return (ticks<0) ? -1:ticks;

}

#if 0
bug
Data8 initTimer0s(uData8 time_s) {

        if (tms==0) {
                initTimer0ms(50);
                ticks = (time_s*ONESECOND);
        } else {
                ticks = (time_s*(SECOND/tms));
        }

        return (ticks<0) ? -1:ticks;

```

```
}
#endif

/**
 * @brief deactivate the timer and check if the count_ticks
 *   variable has already expired. If it expried, i.e.
 *   it elapsed one second then it calls the printValue()
 *   in order to print the sensor data.
 */

#if (TARGET)
static void system_tick(void) interrupt 1
{
   TR0 = 0;              /* stop timer0 (TCON register) */
   TH0 = THIGH;
   TL0 = TLOW;
   timerms_interrupt();
   if (count_ticks++ == ticks){  /* checks if one second has expired */

      count_ticks=0;
      timers_interrupt();

    }

   TR0 = 1;                    /* enable timer0 count (TCON register) */

}
#endif

#if (VERIFICATION)
static void system_tick(void)
{
        int cycle;

   __CPROVER_assume(oc8051_tc.rst==1);

   __CPROVER_assume(oc8051_tc.tr0 == 0);   /* stop timer0 (TCON register) */
   __CPROVER_assume(oc8051_tc.th0==THIGH);
   __CPROVER_assume(oc8051_tc.tl0==TLOW);
   timerms_interrupt();

   if (count_ticks++ == ticks){  /* checks if one second has expired */
          count_ticks=0;
      timers_interrupt();
   }

        oc8051_tc.tr0 = 1;                      /* enable timer0 count (TCON register) */

        for(cycle=0; cycle<bound; cycle++) {
                __CPROVER_assume(oc8051_tc.rst==0);
                next_timeframe();
        }
        assert(oc8051_tc.th0 == THIGH);
        assert(oc8051_tc.tl0 == TLOW);
        assert(oc8051_tc.tr0==1);

}
#endif
```

LISTING A.16: Source code for Pulse Oximeter file pulse/drivers/timer.c

```
/*******************************************************
* File:         timer.h
* Abstract:     Interface of the timer
* Platform:     AT89S8252
* Project       Pulse Oximeter
* Author(s):    Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/

#ifndef _TIMER_H
#define _TIMER_H

/*******************
* INCLUDE FILES    *
*******************/
#include "global.h"
#include "sensor.h"

```

```
#if (TARGET)
#include <8051.h>
#endif

/********************************
* EXPORTED FUNCTIONS PROTOTYPES *
*********************************/
/**
 *   @brief Function used to configure the timer in miliseconds.
 *
 *   @retval  The time in miliseconds.
 */
extern Data8 initTimer0ms(uData8 time_ms);

/**
 *   @brief Function used to configure the timer in seconds.
 *
 *   @retval The time in seconds. Otherwise, -1 is returned.
 */
extern Data8 initTimer0s(uData8 time_s);

/**
 *   @brief Function used to calculate the timer register value
 *
 *   @retval The timer register value
 */
extern uData8 calculateTRegVal(uData8 t);

/**
 *   @brief Function used to calculate the timer high order byte
 *
 *   @retval The timer high order byte
 */
extern uData8 calculateTH(uData8 t);

/**
 *   @brief Function used to calculate the timer low order byte
 *
 *   @retval The timer low order byte
 */
extern uData8 calculateTL(uData8 t);

/**
 *   @brief Procedure that is called according to the timer ticks
 *
 *   @retval void
 */
extern void timers_interrupt(void);

/**
 *   @brief Procedure that is called according to the timer ticks
 *
 *   @retval void
 */
extern void timerms_interrupt(void);

/**
 *   @brief Procedure to initialize the timer
 *
 *   @retval void
 */
extern void initTimers(void);

#endif /* _TIMER_H */
```

LISTING A.17: Source code for Pulse Oximeter file pulse/drivers/timer.h

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File:         LCD.a51
; Abstract:     Functions to access the LCD in Assembly
; Platform:     AT89S8252
; Project       Pulse Oximeter
; Author(s):    Lucas Cordeiro
; Copyright (C)2007 DCC, Federal University of Amazonas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.globl _write2lcd_PARM_1
.globl _write2lcd

```

```
_write2lcd_PARM_1:
                .ds 1
                .area CSEG

_write2lcd:
                mov a,dpl
                lcall wrcar1
                RET

wrcar1:
                push  DPH
                push  DPL

                mov DPTR,#0x8001
                movx  @DPTR,A
                lcall t5ms

                pop   DPL
                pop   DPH
                ret

t5ms:
                MOV TL0,#0xdf
                MOV TH0,#0xb1
                SETB TR0
;               JNB TF0,$
                CLR TR0
                CLR TF0
                RET

; END OF _write2lcd
```

LISTING A.18: Source code for Pulse Oximeter file pulse/drivers/write2lcd.asm

```
/**********************************************************
* File:        buffer.c
* Abstract:    Implementation of the log system
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
**********************************************************/

/************************************************
* INCLUDE FILES                                *
************************************************/
#include<string.h>
#include "log.h"

/************************************************
* LOCAL MACROS                                 *
************************************************/
#define BUFFER_MAX  6400

/*******************************************************************
* STATIC DATA                                                                *
*******************************************************************/
static char  buffer[BUFFER_MAX];     /* BUFFER */

/********************************************************************************
* FUNCTION IMPLEMENTATION                                                                                    *
********************************************************************************/
void initLog(Data8 max) {

        buffer_size = max;
        first = next = 0;
}

Data8 removeLogElement(void) {

#if VERIFICATION
        assert(first>=0);
#endif

        if (next >= 0 && first < buffer_size) {
                first++;
                return buffer[first-1];
        }
        else {
```

```
                return LOGERROR;
        }
}

#if 0
bug
Data8 removeLogElement(void) {

        first++;
        return buffer[first-1];
}
#endif

Data8 insertLogElement(Data8 b) {

        if (next < buffer_size & buffer_size > 0) {
                buffer[next] = b;
                next = (next+1)%buffer_size;
#if VERIFICATION
                assert(next<buffer_size);
#endif
        } else {
                return LOGERROR;
        }

        return b;
}

#if 0
bug
void insertLogElement(Data8 b) {

        buffer[next] = b;
        next = (next+1)%buffer_size;
}
#endif

void logm(char *msg) {

        size_t size;
        int i;

        if (msg!=NULL) {
                size = strlen(msg);
#if VERIFICATION
                assert(size>=0);
#endif
        }

        for(i=0; i<size; i++){
                insertLogElement(msg[i]);
        }

}

Data8 getBufferSize(void) {

        if (buffer[0]!='\0') {
#if VERIFICATION
                assert(strlen(buffer)>=0);
#endif
                return strlen(buffer);
        }

}

Data8 sendLog2PC(void)  {

        uData8 i, err=0;
        size_t bsize;

        if (buffer[0]!='\0')
                bsize = strlen(buffer);

        if (bsize<=0){
                err=-1;
        }

        for(i=0; i<bsize; i++){
        #if (TARGET)
```

```
			SBUF=buffer[i];
			while(TI==0);
			TI=0;
		#endif
			logTransferProgress();
		}

		return err;
}
```

LISTING A.19: Source code for Pulse Oximeter file pulse/utils/log.c

```
/*******************************************************
* File:        log.h
* Abstract:    Interface of the log system
* Platform:    AT89S8252
* Project      Pulse Oximeter
* Author(s):   Lucas Cordeiro
* Copyright (C)2007 DCC, Federal University of Amazonas
*******************************************************/

#ifndef _LOG_H
#define _LOG_H

/******************
* INCLUDE FILES    *
*******************/
#include "../drivers/global.h"
#include "../drivers/sensor.h"

/******************
* EXPORTED MACROS  *
*******************/
#define TOKEN   (#)

#ifdef DEBUG
#define debug(parms...) logm(parms);
#else
#define debug(parms...)
#endif

#define LOGERROR 100

static uData8 first;                 /* Pointer to the input buffer   */
static uData8 next;                  /* Pointer to the output pointer */
static Data8  buffer_size;           /* Max amount of elements in the buffer */

/*******************************
* EXPORTED FUNCTIONS PROTOTYPES *
********************************/
/**
*   @brief Procedure used to set the amount of elements in the buffer
*   and initializes the input and output pointers.
*
*   @retval void
*/
extern void initLog(Data8);

/**
*   @brief Procedure used to remove the current element pointed
*   by the output pointer. It just increments the output
*   pointer.
*
*   @retval The buffer element.
*/
extern Data8 removeLogElement(void);

/**
*   @brief Procedure used to insert an element into the position
*   pointed by the input pointer and increments the input
*   pointer.
*
*   @retval void
*/
extern Data8 insertLogElement(Data8);

/**
*   @brief Procedure used to write in the circular buffer
*
```

```
*   @retval void
*/
extern void logm(char *msg);

/**
*   @brief Function used to send the log messages to the PC
*
*   @retval This function returns 0 if the log is sent successfully.
*                       If an error occurs then -1 is returned.
*/
extern Data8 sendLog2PC(void);

/**
*   @brief Function used to get the buffer size of the log messages
*
*   @retval The length of string
*/
extern Data8 getBufferSize(void);

/**
*   @brief Procedure used to provide the status of transferring
*                  the log messages to the PC.
*
*   @retval The progress of the transferring operation.
*/
extern uData8 logTransferProgress(void);

#endif /* _LOG_H */
```

LISTING A.20: Source code for Pulse Oximeter file pulse/utils/log.h

A.3 Pulse Oximiter test harnesses and patches

This section contains the test harnesses used to invoke different portions of the Pulse Oximeter software, to generate a trace for checking against an LTL formula. It is assumed that a monitor function and associated helpers have already been generated for the formula, producing code as exhibited in Listing A.1. I also list here any modifications made to the code base for running the test, including those modifications to deliberately cause the formula to not hold.

A.3.1 *baud_conf*

Formula: $\mathsf{G}(\{\texttt{brate == 1200}\} \implies \mathsf{F}\{\texttt{TH1 == 0xE8}\})$

```
#include <pthread.h>
#include <stdbool.h>

#include "serial.h"

void serial_init(int);
int nondet_int();
int nondet_uint();
bool nondet_bool();

int event_thread_done = 0;

void ltl2ba_start_monitor();
void ltl2ba_finish_monitor();

void *
event_thread(void *unused)
{
        int conf, i;

        while (1) {
```

```
                conf = nondet_int();
                serial_init(conf);
        }

        event_thread_done = 1;
        return NULL;
}

int
main(int argc, char **argv)
{
        pthread_t evt_loop;

        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_thread, NULL);
        __ESBMC_assume(event_thread_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.21: Test harness for baud_conf LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-18 22:17:20.587406271 +0100
@@ -0,0 +1 @@
+![]({brate == 1200} -> <>{TH1 == 0xE8})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/serial.c ./drivers/serial.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/serial.c      2009-01-17 16:27:49.000000000 +0000
+++ ./drivers/serial.c  2011-04-18 22:07:33.696110432 +0100
@@ -41,6 +41,10 @@ extern const struct module_oc8051_uart o
 extern const struct module_oc8051_tc oc8051_tc;
 extern const struct module_oc8051_int oc8051_int;

+#include <stdbool.h>
+int TH1; /* Model config register */
+int brate; /* Stored config val */
+
 /*******************************************************************************
 * FUNCTION IMPLEMENTATION                                                      *
 *******************************************************************************/
@@ -78,17 +82,25 @@ void serial_init(uData8 baudRate){

        int cycle;
        unsigned char scon_test,wr_addr_bit;
+       uData8 ret;

-#if (TARGET)
+brate = baudRate;

+//#if (TARGET)
+#if 0
        SCON = 0x50;            /* SCON mode 1, 8-bit UART */
        TMOD = 0x20;            /* TMOD: timer 1, mode 2, 8-bit automatic reload */
        TR1 = 1;                        /* TR1: enable timer 1 */
        IE = 0x90;                      /* enable serial interruption */
-       TH1 = calculateTimerVal(baudRate);
-
 #endif
+//     TH1 = calculateTimerVal(baudRate);
+       ret = calculateTimerVal(baudRate);
+TH1 = ret;

+return;
+
+
+#if 0
 #if (VERIFICATION)

        __CPROVER_assume(oc8051_uart.scon=0x50);
@@ -128,6 +140,7 @@ void serial_init(uData8 baudRate){
        assert(oc8051_int.ie==0x90);

 #endif
+#endif


 }
```

LISTING A.22: Patch applied when verifying the baud_conf property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-18 22:19:29.985311973 +0100
@@ -0,0 +1 @@
+![]({brate == 1200} -> <>{TH1 == 0xE8})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/serial.c ./drivers/serial.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/serial.c      2009-01-17 16:27:49.000000000 +0000
+++ ./drivers/serial.c  2011-04-18 22:19:56.923626811 +0100
@@ -41,6 +41,10 @@ extern const struct module_oc8051_uart o
 extern const struct module_oc8051_tc oc8051_tc;
 extern const struct module_oc8051_int oc8051_int;

+#include <stdbool.h>
+int TH1; /* Model config register */
+int brate; /* Stored config val */
+
 /*****************************************************************************
 * FUNCTION IMPLEMENTATION                                                                        *
 *****************************************************************************/
@@ -52,7 +56,7 @@ uData8 calculateTimerVal(uData8 BR){

         case br1200:
             timerVal = reg1200;
-            break;
+//           break;
         case br2400:
             timerVal = reg2400;
             break;
@@ -78,17 +82,25 @@ void serial_init(uData8 baudRate){

        int cycle;
        unsigned char scon_test,wr_addr_bit;
+       uData8 ret;

-#if (TARGET)
+brate = baudRate;

+//#if (TARGET)
+#if 0
        SCON = 0x50;            /* SCON mode 1, 8-bit UART */
        TMOD = 0x20;            /* TMOD: timer 1, mode 2, 8-bit automatic reload */
        TR1 = 1;                        /* TR1: enable timer 1 */
        IE = 0x90;                      /* enable serial interruption */
-       TH1 = calculateTimerVal(baudRate);
-
 #endif
+//     TH1 = calculateTimerVal(baudRate);
+       ret = calculateTimerVal(baudRate);
+TH1 = ret;

+return;
+
+
+#if 0
 #if (VERIFICATION)

        __CPROVER_assume(oc8051_uart.scon=0x50);
@@ -128,6 +140,7 @@ void serial_init(uData8 baudRate){
        assert(oc8051_int.ie==0x90);

 #endif
+#endif


 }
```

LISTING A.23: Patch applied to inject a bug that violates the baud_conf property

A.3.2 *keyb_start*

Formula: $\mathsf{G}(\{\texttt{the\_key == 1}\} \implies \mathsf{F}\{\texttt{command == 1}\})$

```c
#include <pthread.h>
#include <stdbool.h>

#include "keyboard.h"

void initMenuApp(void);
void timerms_interrupt(void);
int nondet_uint();
bool nondet_bool();

int event_thread_done = 0;

void *
event_thread(void *unused)
{
        int key;

        while (1) {
                key = nondet_int() % 8;
                key = 1 << key;

                checkPressedButton(key);
        }

        event_thread_done = 1;
        pthread_exit(NULL);
}

int
main(int argc, char **argv)
{
        pthread_t evt_loop;

        initMenuApp();
        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_thread, NULL);
        __ESBMC_assume(event_thread_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.24: Test harness for keyb_start LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/keyboard.c ./drivers/keyboard.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/keyboard.c    2009-01-17 16:22:11.000000000 +0000
+++ ./drivers/keyboard.c        2011-04-18 22:45:12.122694746 +0100
@@ -21,10 +21,12 @@ enum Key_State {START=BIT0, STOP=BIT1, E
 /********************************************************************************
 * FUNCTION IMPLEMENTATION                                                                                     *
 *********************************************************************************/
+int the_key;
+int command=0;

 Data8 checkPressedButton(uData8 Key){

-    int command=0;
+    the_key = Key;

     switch(Key){

diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-18 21:30:42.025332645 +0100
@@ -0,0 +1 @@
+![]({the_key == 1} -> <>{command == 1})
```

LISTING A.25: Patch applied when verifying the keyb_start property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/keyboard.c ./drivers/keyboard.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/keyboard.c    2009-01-17 16:22:11.000000000 +0000
```

```
+++ ./drivers/keyboard.c        2011-04-18 22:50:38.572144877 +0100
@@ -21,14 +21,16 @@ enum Key_State {START=BIT0, STOP=BIT1, E
 /******************************************************************************
 * FUNCTION IMPLEMENTATION                                                     *
 ******************************************************************************/
+int the_key;
+int command=0;

 Data8 checkPressedButton(uData8 Key){

-    int command=0;
+    the_key = Key;

     switch(Key){

-             case START:            command=startButton;    break;
+             case START:            command=startButton;  //  break;
              case STOP:             command=stopButton;     break;
              case UP:        command=upButton;       break;
              case DOWN:      command=downButton;     break;
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-18 22:50:27.730827266 +0100
@@ -0,0 +1 @@
+![]({the_key == 1} -> <>{command == 1})
```

LISTING A.26: Patch applied to inject a bug that violates the keyb_start property

A.3.3 *serial_rx*

Formula: $\mathsf{G}((\{\texttt{p\_inDat == 1}\} \vee \{\texttt{flag2 == 1}\}) \implies \mathsf{F}\{\texttt{flag1 == 1}\})$

```
#include <pthread.h>
#include <stdbool.h>

#include "serial.h"

extern bool p_contCHK2_condition, p_sensor_byte, q_flag1_on;

void serial_init(int);
void initSensor(void);
void initStatus(void);
void collectData(int sensorByte);
int nondet_int();
int nondet_uint();
bool nondet_bool();
void __ESBMC_atomic_begin();
void __ESBMC_atomic_end();

int event_thread_done = 0;

int p_sensorData;

void *
event_thread(void *unused)
{
        int i;

        while (1) {
                p_sensorData = nondet_int();
                collectData(p_sensorData);
        }

        event_thread_done = 1;
        return NULL;
}

int
main(int argc, char **argv)
{
        pthread_t evt_loop;

//      initSensor(); /* Causes linking against nonexistant foo it seems */
        initStatus();
```

```
        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_thread, NULL);
        __ESBMC_assume(event_thread_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.27: Test harness for serial_rx LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-19 11:16:36.764988737 +0100
@@ -0,0 +1 @@
+![]((({p_sensorData == 1} || {flag2 == 1})) -> <>{flag1 == 1})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/sensor.c ./drivers/sensor.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/sensor.c      2009-01-17 16:36:12.000000000 +0000
+++ ./drivers/sensor.c  2011-04-18 22:31:42.817460070 +0100
@@ -22,9 +22,10 @@ static void fillArrays(Data8 rawData, uD
  * STATIC DATA                                                                                     *
  ******************************************************************/
 static Data8 itr=0, checkSum[SIZEOFFRAME];
-static uData8 contPos=0, contCHK=0, flag1=FALSE, flag2=FALSE, frame=0;
+uData8 contPos=0, contCHK=0, flag1=FALSE, flag2=FALSE, frame=0;
 static char serialData[MAXNUMOFBYTE];
 static char tmp[10];
 static uData8 srev;    /* Oximeter Firmware Revision Level */
@@ -106,6 +107,7 @@ void initStatus(void){
   *   this function sums three HR or SpO2 values and
   *   divides them by the amount was read.
   */
+#if 0
 uData8 showAverage(Data8 *sensorData){

        Data8 i=0, sensorValue=0, numElements=0, aux=0;
@@ -132,6 +134,7 @@ uData8 showAverage(Data8 *sensorData){
        return aux;

 }
+#endif

 uData8 signalInverter(Data8 signal) {
```

LISTING A.28: Patch applied when verifying the serial_rx property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl ./drivers/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/ltl   1970-01-01 01:00:00.000000000 +0100
+++ ./drivers/ltl       2011-04-19 11:17:27.926776024 +0100
@@ -0,0 +1 @@
+![]((({p_sensorData == 1} || {flag2 == 1})) -> <>{flag1 == 1})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/sensor.c ./drivers/sensor.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/drivers/sensor.c      2009-01-17 16:36:12.000000000 +0000
+++ ./drivers/sensor.c  2011-04-18 22:38:46.363952134 +0100
@@ -22,9 +22,10 @@ static void fillArrays(Data8 rawData, uD
  * STATIC DATA                                                                                     *
  ******************************************************************/
 static Data8 itr=0, checkSum[SIZEOFFRAME];
-static uData8 contPos=0, contCHK=0, flag1=FALSE, flag2=FALSE, frame=0;
+uData8 contPos=0, contCHK=0, flag1=FALSE, flag2=FALSE, frame=0;
 static char serialData[MAXNUMOFBYTE];
 static char tmp[10];
 static uData8 srev;    /* Oximeter Firmware Revision Level */
@@ -106,6 +107,7 @@ void initStatus(void){
  *   this function sums three HR or SpO2 values and
  *   divides them by the amount was read.
  */
+#if 0
 uData8 showAverage(Data8 *sensorData){

        Data8 i=0, sensorValue=0, numElements=0, aux=0;
@@ -132,6 +134,7 @@ uData8 showAverage(Data8 *sensorData){
        return aux;

 }
+#endif

```

```
 uData8 signalInverter(Data8 signal) {

@@ -177,6 +180,8 @@ uData8 checkValidBytes(Data8 *chBytes) {
  *   in an array that will be used further in order to fill in
  *   other arrays.
  */
+#include <stdbool.h>
+bool nondet_bool();
 void collectData(Data8 sensorByte) {

        int chkerr, i;
@@ -200,7 +205,11 @@ void collectData(Data8 sensorByte) {
                else if ((SYNC&sensorByte) == TRUE || flag1 == TRUE) {
                        checkSum[contCHK]=sensorByte;
                        contCHK++;
-               }       flag1=TRUE;
+                       if (nondet_bool())
+                               flag1=TRUE;
+                       else
+                               flag1 =FALSE;
+               }
        }
        if(frame==3){
                srev = checkSum[3];
```

LISTING A.29: Patch applied to inject a bug that violates the serial_rx property

A.3.4 *up_btn*

Formula: $\mathsf{G}((\{\texttt{press == 4}\} \wedge \{\texttt{mstate == 1}\}) \implies \mathsf{F}\{\texttt{stime > refstime}\})$

```
#include <pthread.h>
#include <stdbool.h>

#include "menu_app.h"

void initMenuApp(void);
void timerms_interrupt(void);
void setMstate(int s);

int event_loop_done = 0;
extern bool q_stimeHasIncreased;

extern unsigned int stime;
unsigned int ref_stime;

void *
event_thread(void *unused)
{
        int i;

        while (1) {
                /* Set state to some arbitary situation */
                setMstate(SETSAMPLETIME);

                /* Then fire a few intrs */
                /* Reset condition we're checking whenever we send input*/
                ref_stime = stime;
                timerms_interrupt();

        }

        event_loop_done = 1;
        return NULL;
}

int
main(int argc, char **argv)
{
        pthread_t thread, evt_loop;

        initMenuApp();
        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_thread, NULL);
```

```
        __ESBMC_assume(event_loop_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.30: Test harness for up_btn LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl ./apps/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl      1970-01-01 01:00:00.000000000 +0100
+++ ./apps/ltl  2011-04-18 16:00:46.743729860 +0100
@@ -0,0 +1 @@
+![](({pressed_key == 4} && {mstate == 1}) -> <>{stime > ref_stime})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c ./apps/menu_app.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c       2009-01-17 16:32:10.000000000 +0000
+++ ./apps/menu_app.c   2011-04-18 20:36:41.537972365 +0100
@@ -35,6 +35,7 @@
         parm++;             \
         bounce=BOUNCEVAL;   \
     } else {                \
+parm++; /* Disable debouncing for this test */ \
         bounce--;           \
         if (bounce<0) {     \
             bounce=0;       \
@@ -110,12 +111,16 @@ typedef struct {
 /******************************************************************
 * STATIC DATA                                                                    *
 ******************************************************************/
-static uData8 stime, elog, bounce, exists_log;
+uData8 stime, elog, bounce, exists_log;
 static uData8 count_pos, count_elem, count, global_progress, unit_progress;
-static uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
+uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
 showData show;
 static char menuVal[10], opData[AMOUNTOFDATA];

+#include <stdbool.h>
+
 /*******************************************************************************
 * FUNCTION IMPLEMENTATION                                                                          *
 *******************************************************************************/
@@ -150,6 +155,13 @@ void initMenuApp (void) {
        unit_progress=0;
 }

+void setMstate(int s)
+{
+
+       mstate = s;
+       return;
+}
+
 uData8 selectItem(void) {

        if (enable_buttons) {
@@ -253,6 +265,7 @@ uData8 KeyUp(void) {
                 SETINC(stime);
                               result=stime;
                 break;
+#if 0
             case SETLOG:
                 SETNEG(elog);
                               result=elog;
@@ -307,12 +320,15 @@ uData8 KeyUp(void) {
                 break;
                        default:
                               result=-1;
+#endif
         }
     }

+#if 0
 #if VERIFICATION
        assert(result>=0);
 #endif
+#endif

        return result;
 }
```

```
@@ -744,17 +760,20 @@ void timerms_interrupt(void){

 #if (TARGET)
     keys=P1;
+    pressed_key = checkPressedButton(keys);
 #endif

 #if VERIFICATION
-       assert(keys>=0);
+       pressed_key = upButton;
+       //assert(keys>=0);
 #endif

-    pressed_key = checkPressedButton(keys);
-
     if(pressed_key>0){

         switch(pressed_key){
+#if 0
                case startButton:
                                startApp();
                        break;
@@ -764,15 +783,18 @@ void timerms_interrupt(void){
                case emptyButton:
                        empty();
                        break;
+#endif
                case upButton:
                        KeyUp();
                        break;
+#if 0
                case downButton:
                        KeyDown();
                        break;
                case selectButton:
                        selectItem();
                        break;
+#endif
         }

     }
```

LISTING A.31: Patch applied when verifying the up_btn property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl ./apps/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl      1970-01-01 01:00:00.000000000 +0100
+++ ./apps/ltl  2011-04-18 20:16:08.187119563 +0100
@@ -0,0 +1 @@
+![](({pressed_key == 4} && {mstate == 1}) -> <>{stime > ref_stime})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c ./apps/menu_app.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c       2009-01-17 16:32:10.000000000 +0000
+++ ./apps/menu_app.c   2011-04-18 20:16:08.187119563 +0100
@@ -35,6 +35,7 @@
         parm++;             \
         bounce=BOUNCEVAL;   \
     } else {                \
+/*parm++;*/ /* Disable debouncing for this test */ \
         bounce--;           \
         if (bounce<0) {     \
             bounce=0;       \
@@ -110,12 +111,16 @@ typedef struct {
 /*****************************************************************
 * STATIC DATA                                                                      *
 *****************************************************************/
-static uData8 stime, elog, bounce, exists_log;
+uData8 stime, elog, bounce, exists_log;
 static uData8 count_pos, count_elem, count, global_progress, unit_progress;
-static uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
+uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
 showData show;
 static char menuVal[10], opData[AMOUNTOFDATA];

+#include <stdbool.h>
+
 /******************************************************************************
 * FUNCTION IMPLEMENTATION                                                                          *
 ******************************************************************************/
@@ -150,6 +155,13 @@ void initMenuApp (void) {
        unit_progress=0;
```

```
 }

+void setMstate(int s)
+{
+
+       mstate = s;
+       return;
+}
+
 uData8 selectItem(void) {

        if (enable_buttons) {
@@ -253,6 +265,7 @@ uData8 KeyUp(void) {
                SETINC(stime);
                               result=stime;
                break;
+#if 0
            case SETLOG:
                SETNEG(elog);
                               result=elog;
@@ -307,12 +320,15 @@ uData8 KeyUp(void) {
                break;
                       default:
                               result=-1;
+#endif
         }
     }

+#if 0
 #if VERIFICATION
        assert(result>=0);
 #endif
+#endif

        return result;
 }
@@ -744,17 +760,20 @@ void timerms_interrupt(void){

 #if (TARGET)
     keys=P1;
+    pressed_key = checkPressedButton(keys);
 #endif

 #if VERIFICATION
-       assert(keys>=0);
+       pressed_key = upButton;
+       //assert(keys>=0);
 #endif

-    pressed_key = checkPressedButton(keys);
-
     if(pressed_key>0){

         switch(pressed_key){
+#if 0
               case startButton:
                               startApp();
                       break;
@@ -764,15 +783,18 @@ void timerms_interrupt(void){
               case emptyButton:
                       empty();
                       break;
+#endif
               case upButton:
                       KeyUp();
                       break;
+#if 0
               case downButton:
                       KeyDown();
                       break;
               case selectButton:
                       selectItem();
                       break;
+#endif
         }

     }
```

LISTING A.32: Patch applied to inject a bug that violates the up_btn property

A.3.5 *start_btn*

Formula: $\mathsf{G}((\neg\{\texttt{press == 1}\} \wedge \mathsf{F}\{\texttt{press == 1}\}) \implies \mathsf{F}\{\texttt{q\_startCall}\})$

```
#include <pthread.h>
#include <stdbool.h>

void initMenuApp(void);
void timerms_interrupt(void);
bool nondet_bool();

void ltl2ba_start_monitor();
void ltl2ba_finish_monitor();

int event_loop_done = 0;

int
event_loop(void *dummy)
{
        int i;

__ESBMC_yield();
        while (1) {
                timerms_interrupt();
        }

        event_loop_done = 1;
        return 0;
}

int
main(int argc, char **argv)
{
        pthread_t thread, evt_loop;

        initMenuApp();

        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_loop, NULL);
        __ESBMC_assume(event_loop_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.33: Test harness for start_btn LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl ./apps/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl      1970-01-01 01:00:00.000000000 +0100
+++ ./apps/ltl  2011-04-19 16:03:20.608695728 +0100
@@ -0,0 +1 @@
+![]((((!{pressed_key == 1})) && <>{pressed_key == 1}) -> <>{q_startAppCalled})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c ./apps/menu_app.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c       2009-01-17 16:32:10.000000000 +0000
+++ ./apps/menu_app.c   2011-04-19 14:55:17.413189427 +0100
@@ -112,10 +112,12 @@ typedef struct {
 *******************************************************************/
 static uData8 stime, elog, bounce, exists_log;
 static uData8 count_pos, count_elem, count, global_progress, unit_progress;
-static uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
+uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
 showData show;
 static char menuVal[10], opData[AMOUNTOFDATA];

+#include <stdbool.h>
+
 /*****************************************************************************
 * FUNCTION IMPLEMENTATION                                                                    *
 *****************************************************************************/
@@ -207,8 +209,10 @@ uData8 selectItem(void) {
        return mstate;
 }

+bool q_startAppCalled;
 static void startApp(void) {

+q_startAppCalled = true;
```

```
        if (exists_log) {
                mstate=CONCABLE;
                if (connect_cable){
@@ -223,6 +227,7 @@ static void startApp(void) {
        enable_buttons=FALSE;
        setCountElem();
        }
+q_startAppCalled = false;
 }

 static void empty(void) {
@@ -737,27 +742,29 @@ uData8 logTransferProgress(void) {

        return global_progress;
 }
-
+bool nondet_bool();
 void timerms_interrupt(void){

     uData8 keys=0x00; /* no key pressed */

 #if (TARGET)
     keys=P1;
+    pressed_key = checkPressedButton(keys);
 #endif

 #if VERIFICATION
-       assert(keys>=0);
+       pressed_key = startButton;
+        //this indicates that startButton has been pressed
+       //assert(keys>=0);
 #endif

-    pressed_key = checkPressedButton(keys);
-
     if(pressed_key>0){

         switch(pressed_key){
                case startButton:
                                startApp();
                        break;
+#if 0
                case stopButton:
                        stopApp();
                        break;
@@ -773,9 +780,12 @@ void timerms_interrupt(void){
                case selectButton:
                        selectItem();
                        break;
+#endif
         }

     }
+
+    pressed_key = 0;
 }

 void setCountElem(void){
```

LISTING A.34: Patch applied when verifying the start_btn property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl ./apps/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/ltl      1970-01-01 01:00:00.000000000 +0100
+++ ./apps/ltl  2011-04-18 15:13:09.434791426 +0100
@@ -0,0 +1 @@
+![]((((!{pressed_key == 1})) && <>{pressed_key == 1}) -> <>{q_startAppCalled})
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c ./apps/menu_app.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/apps/menu_app.c       2009-01-17 16:32:10.000000000 +0000
+++ ./apps/menu_app.c   2011-04-18 15:37:46.437078315 +0100
@@ -112,10 +112,12 @@ typedef struct {
 *****************************************************************/
 static uData8 stime, elog, bounce, exists_log;
 static uData8 count_pos, count_elem, count, global_progress, unit_progress;
-static uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
+uData8 pressed_key, mstate, amount, enable_buttons, log2pc, connect_cable;
 showData show;
 static char menuVal[10], opData[AMOUNTOFDATA];

+#include <stdbool.h>
```

```
+
 /*****************************************************************************
 * FUNCTION IMPLEMENTATION                                                                                *
 *****************************************************************************/
@@ -207,8 +209,10 @@ uData8 selectItem(void) {
        return mstate;
 }

+bool q_startAppCalled;
 static void startApp(void) {

+//q_startAppCalled = true;
        if (exists_log) {
                mstate=CONCABLE;
                if (connect_cable){
@@ -223,6 +227,7 @@ static void startApp(void) {
        enable_buttons=FALSE;
        setCountElem();
        }
+q_startAppCalled = false;
 }

 static void empty(void) {
@@ -737,27 +742,29 @@ uData8 logTransferProgress(void) {

        return global_progress;
 }
-
+bool nondet_bool();
 void timerms_interrupt(void){

     uData8 keys=0x00; /* no key pressed */

 #if (TARGET)
     keys=P1;
+    pressed_key = checkPressedButton(keys);
 #endif

 #if VERIFICATION
-       assert(keys>=0);
+       pressed_key = startButton;
+        //this indicates that startButton has been pressed
+       //assert(keys>=0);
 #endif

-    pressed_key = checkPressedButton(keys);
-
     if(pressed_key>0){

         switch(pressed_key){
                case startButton:
                                startApp();
                        break;
+#if 0
                case stopButton:
                        stopApp();
                        break;
@@ -773,9 +780,12 @@ void timerms_interrupt(void){
                case selectButton:
                        selectItem();
                        break;
+#endif
         }

     }
+
+    pressed_key = 0;
 }

 void setCountElem(void){
```

LISTING A.35: Patch applied to inject a bug that violates the start_btn property

A.3.6 *buflim*

Formula: G({`buffer_size != 0`} $\implies$ {`next < buffer_size`})

```c
#include <pthread.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

extern bool q_bufferOverflowCond;
extern void initLog(int sz);
extern void logm(char *msg);
int nondet_uint();
int nondet_int();
bool nondet_bool();
extern int getchar();
void __ESBMC_assume(bool x);

int event_loop_done = 0;

void *
event_thread(void *unused)
{
        int sz = nondet_int();
        int strsz;
        char *str;

        initLog(sz);

        while (1) {
                int i;
                strsz = nondet_int();
                str = malloc(strsz);
                for (i = 0; i < strsz; i++)
                        str[i] = getchar();

                str[strsz-1] = 0;

                logm(str);

                free(str);
        }

        event_loop_done = 1;
        return NULL;
}

int
main(int argc, char **argv)
{
        pthread_t thread, evt_loop;

        initMenuApp();
        ltl2ba_start_monitor();
        pthread_create(&evt_loop, NULL, event_thread, NULL);
        __ESBMC_assume(event_loop_done != 0);
        ltl2ba_finish_monitor();

        return 0;
}
```

LISTING A.36: Test harness for buflim LTL property

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.c ./utils/log.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.c   2009-01-17 17:01:24.000000000 +0000
+++ ./utils/log.c       2011-04-12 12:04:32.095896808 +0100
@@ -26,12 +26,18 @@ static char  buffer[BUFFER_MAX];     /*
 /****************************************************************************
 * FUNCTION IMPLEMENTATION                                                                  *
 ****************************************************************************/
+
+#include <stdbool.h>
+
 void initLog(Data8 max) {

        buffer_size = max;
        first = next = 0;
```

```
 }

+#if 0
 Data8 removeLogElement(void) {

 #if VERIFICATION
@@ -46,6 +52,7 @@ Data8 removeLogElement(void) {
                return LOGERROR;
        }
 }
+#endif

 #if 0
 bug
@@ -61,6 +68,7 @@ Data8 insertLogElement(Data8 b) {
        if (next < buffer_size & buffer_size > 0) {
                buffer[next] = b;
                next = (next+1)%buffer_size;
 #if VERIFICATION
                assert(next<buffer_size);
 #endif
@@ -98,6 +106,7 @@ void logm(char *msg) {

 }

+#if 0
 Data8 getBufferSize(void) {

        if (buffer[0]!='\0') {
@@ -108,7 +117,8 @@ Data8 getBufferSize(void) {
        }

 }
-
+#endif
+#if 0
 Data8 sendLog2PC(void) {

        uData8 i, err=0;
@@ -132,3 +142,4 @@ Data8 sendLog2PC(void)      {

        return err;
 }
+#endif
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/ltl ./utils/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/ltl     1970-01-01 01:00:00.000000000 +0100
+++ ./utils/ltl 2011-04-18 21:05:33.400092859 +0100
@@ -0,0 +1 @@
+![]({buffer_size != 0} -> {next < buffer_size})
```

LISTING A.37: Patch applied when verifying the buflim property holds

```
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.c ./utils/log.c
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.c   2009-01-17 17:01:24.000000000 +0000
+++ ./utils/log.c       2011-04-18 21:41:33.042403103 +0100
@@ -26,12 +26,16 @@ static char  buffer[BUFFER_MAX];     /*
 /*******************************************************************************
 * FUNCTION IMPLEMENTATION                                                                         *
 *******************************************************************************/
+
+#include <stdbool.h>
+
 void initLog(Data8 max) {

        buffer_size = max;
        first = next = 0;
 }

+#if 0
 Data8 removeLogElement(void) {

 #if VERIFICATION
@@ -46,6 +50,7 @@ Data8 removeLogElement(void) {
                return LOGERROR;
        }
 }
+#endif

 #if 0
```

```
 bug
@@ -60,7 +65,9 @@ Data8 insertLogElement(Data8 b) {

        if (next < buffer_size & buffer_size > 0) {
                buffer[next] = b;
-               next = (next+1)%buffer_size;
+//             next = (next+1)%buffer_size;
+               next %= buffer_size;
+               next += 1; /* Breaking */
 #if VERIFICATION
                assert(next<buffer_size);
 #endif
@@ -98,6 +105,7 @@ void logm(char *msg) {

 }

+#if 0
 Data8 getBufferSize(void) {

        if (buffer[0]!='\0') {
@@ -108,7 +116,8 @@ Data8 getBufferSize(void) {
        }

 }
-
+#endif
+#if 0
 Data8 sendLog2PC(void) {

        uData8 i, err=0;
@@ -132,3 +141,4 @@ Data8 sendLog2PC(void)      {

        return err;
 }
+#endif
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.h ./utils/log.h
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/log.h   2009-01-12 22:11:34.000000000 +0000
+++ ./utils/log.h       2011-04-18 21:42:29.794835893 +0100
@@ -30,8 +30,8 @@
 #define LOGERROR 100

 static uData8 first;                /* Pointer to the input buffer   */
-static uData8 next;                 /* Pointer to the output pointer */
-static Data8  buffer_size;          /* Max amount of elements in the buffer */
+uData8 next;                 /* Pointer to the output pointer */
+Data8  buffer_size;          /* Max amount of elements in the buffer */

 /********************************
 * EXPORTED FUNCTIONS PROTOTYPES *
diff -r -N -E -b -up /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/ltl ./utils/ltl
--- /home/jmorse/Downloads/tmp/pulse_oximeter/src/utils/ltl     1970-01-01 01:00:00.000000000 +0100
+++ ./utils/ltl 2011-04-18 21:13:07.997528631 +0100
@@ -0,0 +1 @@
+![]({buffer_size != 0} -> {next < buffer_size})
```

LISTING A.38: Patch applied to inject a bug that violates the buflim property

A.4 Bicycle Computer

```
#include <time.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include <pthread.h>

#include <sys/time.h>

pthread_t ltl2ba_start_monitor(void);
void ltl2ba_finish_monitor(pthread_t t);

enum statet {
        trip_state = 0, speed_state = 1, total_state = 2, time_state = 3
};
```

```c
pthread_mutex_t cycle_dist_lock;
uint64_t cycle_distance_m = 0;
uint64_t total_cycle_distance_m = 0;
enum statet cur_state = 0;
struct timeval starttime;
unsigned int display = 0;

void __ESBMC_really_atomic_begin();
void __ESBMC_really_atomic_end();

int
fprintf(const char *fmt, ...)
{
	display = 1;
	display = 0;
}

static unsigned int
state2time(enum statet thestate)
{

	switch (thestate) {
	case trip_state:
		return 200;
	case speed_state:
		return 100;
	case total_state:
		return 500;
	case time_state:
		return 1000;
	}
}

void *
printing_thread(void *dummy)
{
	enum statet captured_state;
	uint64_t captured_distance, s_since, captured_total_distance;
	struct timeval captured_time, now;
	struct timespec time_to_sleep;
	double speed;

	while (true) {

		pthread_mutex_lock(&cycle_dist_lock);
		__ESBMC_really_atomic_begin();
		captured_state = cur_state;
		captured_distance = cycle_distance_m;
		captured_total_distance = total_cycle_distance_m;
		captured_time = starttime;
		__ESBMC_really_atomic_end();
		pthread_mutex_unlock(&cycle_dist_lock);

		gettimeofday(&now, NULL);

		switch (captured_state) {
		case trip_state:
			/* Mileage */
			fprintf(stderr, "Mileage: %llum\n", captured_distance);
			break;
		case speed_state:
			s_since = now.tv_sec - captured_time.tv_sec;
			if (s_since != 0)
				speed = (double)captured_distance / (double)s_since;
			else
				speed = 0;

			fprintf(stderr, "Speed: %f M/S\n", speed);
			break;
		case total_state:
			fprintf(stderr, "Total Mileage: %llum\n",
					captured_total_distance);
			break;
		case time_state:
			s_since = now.tv_sec - captured_time.tv_sec;
			fprintf(stderr, "Time: %d seconds\n", s_since);
			break;
		}
```

```
                time_to_sleep.tv_nsec = state2time(captured_state) * 1000000;
                time_to_sleep.tv_sec = time_to_sleep.tv_nsec / 1000000000;
                time_to_sleep.tv_nsec %= 1000000000;
                nanosleep(&time_to_sleep, NULL);
        }

        return NULL;
}

void *
cycling_thread(void *dummy)
{
        struct timespec time_to_sleep;

        time_to_sleep.tv_sec = 0;
        time_to_sleep.tv_nsec = 100000000;

        // Follow existing progress formula, not defined in spec

        while (true) {
                nanosleep(&time_to_sleep, NULL);
                if ((rand() % 3) == 0) {
                        pthread_mutex_lock(&cycle_dist_lock);
                        __ESBMC_really_atomic_begin();
                        cycle_distance_m++;
                        total_cycle_distance_m++;
                        __ESBMC_really_atomic_end();
                        pthread_mutex_unlock(&cycle_dist_lock);
                }
        }

        return NULL;
}

int input;
void __ESBMC_switch_to_monitor(void);

int
main()
{
        pthread_t cycling, printing;

        pthread_t monitor = ltl2ba_start_monitor();
        gettimeofday(&starttime, NULL);
        pthread_mutex_init(&cycle_dist_lock, NULL);
        __ESBMC_atomic_begin();
        pthread_create(&cycling, NULL, cycling_thread, NULL);
        pthread_create(&printing, NULL, printing_thread, NULL);
        __ESBMC_atomic_end();

        do {
                printf("Cycling options:\n");
                printf("1) Reset button\n");
                printf("2) Mode button\n");
                printf("3) Quit\n");
                //scanf("%d", &input);
                int face = nondet_int();
                input = face;

                switch (input) {
                default:
                        printf("Not a valid input\n");
                        break;
                case 1:
                        pthread_mutex_lock(&cycle_dist_lock);
                        __ESBMC_really_atomic_begin();
                        gettimeofday(&starttime, NULL);
                        cycle_distance_m = 0;
                        __ESBMC_really_atomic_end();
                        pthread_mutex_unlock(&cycle_dist_lock);
                        break;
                case 2:
                        pthread_mutex_lock(&cycle_dist_lock);
                        __ESBMC_really_atomic_begin();
                        cur_state = (cur_state + 1) % 4;
                        __ESBMC_really_atomic_end();
                        pthread_mutex_unlock(&cycle_dist_lock);
                        break;
                case 3:
                        goto out; // 100% legitimate use of goto
```

```
                }
        } while (1);

out:
        // Clear up

        ltl2ba_finish_monitor(monitor);

        exit(0);
}
```

LISTING A.39: Source code for example bicycle computer, multithreaded LTL test program

Appendix B

Concurrency optimisation results

This appendix contains the results of the multithreaded optimisation test runs, as discussed in Chapter 5. The first table below identifies the longest running test run for a particular benchmark, while all the other tables show a comparison between ESBMC running without any optimisation, and one with. Tables are shown twice, one with rows ordered by the performance gain, the other ordered by testname to aid comparisons. Aside from the first table, test names are truncated to 30 characters to aid layout (all remain unique).

The comparison tables have one row per test run, identified by its testname, and several statistics regarding the performance of the run, with and without the optimisation. The 'Unopt' column contains the amount of time (in seconds) consumed by the unoptimised version of ESBMC for this test run, and the (variously named) following column contains the amount of time taken with the optimisation. The "Pct diff" column contains that percentage difference between the unoptimised and optimised version.

Following this are two more columns, "UI" and "OI". UI contains the number of interleavings that are explored with the unoptimised version of ESBMC, and OI the number that are explored if the relevant optimisation is enabled. This column is not meaningful for the incremental solving results, as it aims to improve the speed at which states are verified, rather than reducing the amount of state space.

A number of test runs crash due to running out of memory—these are marked with "Crash" in the column corresponding to the time consumed, and other performance statistics are marked N/A.

Directory	Testname	Time (s)	UB	CB	#I
pthread-ext	01_inc_true.i	5061	3	7	1286217
pthread-ext	02_inc_cas_true.i	637	3	5	112698
pthread-ext	03_incdec_true.i	1746	3	3	45261
pthread-ext	04_incdec_cas_true.i	1814	4	2	15314
pthread-ext	05_tas_true.i	5529	2	6	558538

pthread-ext	06_ticket_true.i	402	4	3	124130
pthread-ext	07_rand_true.i	7318	4	3	120356
pthread-ext	08_rand_cas_true.i	4026	3	4	79716
pthread-ext	09_fmaxsym_true.i	679	2	6	76694
pthread-ext	10_fmaxsym_cas_true.i	823	3	3	18567
pthread-ext	11_fmaxsymopt_true.i	6981	3	4	106094
pthread-ext	12_fmaxsymopt_cas_true.i	2318	2	7	53810
pthread-ext	13_unverif_true.i	1327	3	5	181251
pthread-ext	14_spin2003_true.i	1859	4	6	2882659
pthread-ext	15_dekker_true.i	255	5	7	273452
pthread-ext	16_peterson_true.i	4	5	7	9614
pthread-ext	17_szymanski_true.i	4191	5	4	395961
pthread-ext	20_lamport_true.i	2142	2	7	754592
pthread-ext	25_stack_true.i	1273	3	3	105955
pthread-ext	26_stack_cas_true.i	3504	3	3	140503
pthread-ext	27_Boop_simple_vf_false.i	9	5	1	8748
pthread-ext	28_buggy_simple_loop1_vf_false.i	1026	5	7	3817
pthread-ext	29_conditionals_vs_true.i	5306	4	6	173027
pthread-ext	30_Function_Pointer3_vs_true.i	3302	4	3	246714
pthread-ext	31_simple_loop5_vs_true.i	2255	4	3	2042618
pthread-ext	32_pthread5_vs_false.i	8789	1	7	449073
pthread-ext	33_double_lock_p1_vs_true.i	1091	3	3	112830
pthread-ext	34_double_lock_p2_vs_true.i	1878	3	3	132280
pthread-ext	35_double_lock_p3_vs_true.i	8854	5	2	501180
pthread-ext	36_stack_cas_p0_vs_concur_true.i	4603	3	3	114363
pthread-ext	37_stack_lock_p0_vs_concur_true.i	5906	5	2	286164
pthread-ext	38_rand_cas_vs_concur_true.i	2974	5	2	18264
pthread-ext	39_rand_lock_p0_vs_true.i	5883	3	6	421378
pthread-ext	45_monabsex1_vs_true.i	1674	5	4	856860
pthread-ext	47_ticket_lock_hc_backoff_vs_true.i	1083	3	3	406785
pthread-ext	48_ticket_lock_low_contention_vs_true.i	10649	2	6	1355947
pthread-atomic	dekker_true.i	1991	3	7	2278580
pthread	fib_bench_false.i	23	5	7	43830
pthread	fib_bench_longer_false.i	23	5	7	43830
pthread	fib_bench_longer_true.i	23	5	7	43830
pthread	fib_bench_longest_false.i	24	5	7	43830
pthread	fib_bench_longest_true.i	24	5	7	43830
pthread	fib_bench_true.i	23	5	7	43830
pthread-atomic	lamport_true.i	718	5	5	499831

pthread	lazy01_false.i	932	2	7	412364
pthread-atomic	peterson_true.i	439	5	7	674597
pthread-atomic	qrcu_false.i	4966	3	3	112097
pthread-atomic	qrcu_true.i	1484	3	3	112097
pthread	queue_false.i	411	1	5	136484
pthread	queue_ok_true.i	905	2	7	722966
pthread-atomic	read_write_lock_false.i	1006	4	4	841504
pthread-atomic	read_write_lock_true.i	789	4	4	655608
pthread	reorder_2_false.i	998	5	3	160660
pthread	reorder_5_false.i	4318	4	5	3604813
pthread-atomic	scull_true.i	643	2	3	93416
pthread	sigma_false.i	10965	5	5	7265574
pthread	sigma_true.i	11372	5	5	7265574
pthread	singleton_false.i	12562	1	5	1253886
pthread	singleton_true.i	4717	2	5	1253886
pthread	stack_false.i	5243	3	6	357623
pthread	stack_true.i	3321	3	6	298289
pthread	stateful01_false.i	90	3	7	73675
pthread	stateful01_true.i	83	4	7	69241
pthread	sync01_true.i	118	5	7	71687
pthread-atomic	szymanski_true.i	2005	1	7	2591205
pthread-atomic	time_var_mutex_true.i	881	2	7	443694
pthread	twostage_3_false.i	566	2	6	400616

TABLE B.1: The 67 tests and bound configurations used to evaluate optimisations. UB signifies the unwinding bound and CB the context switch bound, and #I the number of interleavings generated

Test name	Unopt	SHash	Pct diff	UI	OI
15_dekker_true.i	255	0	0.0000	273452	442
fib_bench_longest_true.i	24	0	0.0000	43830	597
fib_bench_longer_true.i	23	0	0.0000	43830	597
fib_bench_false.i	23	0	0.0000	43830	597
stateful01_true.i	83	0	0.0000	69241	264
16_peterson_true.i	4	0	0.0000	9614	53
fib_bench_true.i	23	0	0.0000	43830	597
fib_bench_longer_false.i	23	0	0.0000	43830	597
20_lamport_true.i	2142	1	0.0467	754592	1324
32_pthread5_vs_false.i	8789	12	0.1365	449073	16682
01_inc_true.i	5061	11	0.2173	1286217	13215

17_szymanski_true.i	4191	14	0.3340	395961	6015
peterson_true.i	439	2	0.4556	674597	2690
dekker_true.i	1991	10	0.5023	2278580	11839
48_ticket_lock_low_contention_	10649	89	0.8358	1355947	71752
time_var_mutex_true.i	881	8	0.9081	443694	8044
02_inc_cas_true.i	637	9	1.4129	112698	4489
31_simple_loop5_vs_true.i	2255	34	1.5078	2042618	50817
lazy01_false.i	932	16	1.7167	412364	14585
13_unverif_true.i	1327	27	2.0347	181251	14416
39_rand_lock_p0_vs_true.i	5883	125	2.1248	421378	61330
stateful01_false.i	90	2	2.2222	73675	1676
14_spin2003_true.i	1859	47	2.5282	2882659	75961
szymanski_true.i	2005	53	2.6434	2591205	60366
lamport_true.i	718	20	2.7855	499831	13539
05_tas_true.i	5529	171	3.0928	558538	53657
fib_bench_longest_false.i	24	1	4.1667	43830	597
45_monabsex1_vs_true.i	1674	88	5.2569	856860	77314
stack_false.i	5243	279	5.3214	357623	94729
queue_ok_true.i	905	52	5.7459	722966	45814
sync01_true.i	118	8	6.7797	71687	3247
29_conditionals_vs_true.i	5306	444	8.3679	173027	73947
stack_true.i	3321	280	8.4312	298289	92010
queue_false.i	411	49	11.9221	136484	28136
twostage_3_false.i	566	68	12.0141	400616	52735
read_write_lock_false.i	1006	184	18.2903	841504	132079
reorder_5_false.i	4318	808	18.7124	3604813	618407
read_write_lock_true.i	789	148	18.7579	655608	106026
08_rand_cas_true.i	4026	783	19.4486	79716	28258
09_fmaxsym_true.i	679	135	19.8822	76694	27046
06_ticket_true.i	402	86	21.3930	124130	38599
47_ticket_lock_hc_backoff_vs_t	1083	329	30.3786	406785	157045
27_Boop_simple_vf_false.i	9	3	33.3333	8748	2976
07_rand_true.i	7318	2504	34.2170	120356	72209
singleton_false.i	12562	6449	51.3374	1253886	825713
reorder_2_false.i	998	599	60.0200	160660	111301
qrcu_false.i	4966	3584	72.1708	112097	101055
singleton_true.i	4717	3425	72.6097	1253886	825713
35_double_lock_p3_vs_true.i	8854	6465	73.0178	501180	422207
12_fmaxsymopt_cas_true.i	2318	1772	76.4452	53810	48948

sigma_true.i	11372	8794	77.3303	7265574	5450284
11_fmaxsymopt_true.i	6981	5535	79.2866	106094	100234
36_stack_cas_p0_vs_concur_true	4603	3663	79.5785	114363	110942
sigma_false.i	10965	8823	80.4651	7265574	5450284
30_Function_Pointer3_vs_true.i	3302	2704	81.8898	246714	219658
25_stack_true.i	1273	1099	86.3315	105955	98719
qrcu_true.i	1484	1330	89.6226	112097	101055
34_double_lock_p2_vs_true.i	1878	1718	91.4803	132280	119041
10_fmaxsym_cas_true.i	823	776	94.2892	18567	18567
26_stack_cas_true.i	3504	3369	96.1473	140503	134069
38_rand_cas_vs_concur_true.i	2974	2886	97.0410	18264	17193
03_incdec_true.i	1746	1704	97.5945	45261	44997
37_stack_lock_p0_vs_concur_tru	5906	5768	97.6634	286164	259645
33_double_lock_p1_vs_true.i	1091	1071	98.1668	112830	98073
04_incdec_cas_true.i	1814	1875	103.3627	15314	15314
scull_true.i	643	675	104.9767	93416	88300
28_buggy_simple_loop1_vf_false	1026	1094	106.6277	3817	3817

TABLE B.2: Performance differences between unoptimised ESBMC and ESBMC with state hashing enabled, ordered by performance improvement. See Appendix B for table format

Test name	Unopt	SHash	Pct diff	UI	OI
01_inc_true.i	5061	11	0.2173	1286217	13215
02_inc_cas_true.i	637	9	1.4129	112698	4489
03_incdec_true.i	1746	1704	97.5945	45261	44997
04_incdec_cas_true.i	1814	1875	103.3627	15314	15314
05_tas_true.i	5529	171	3.0928	558538	53657
06_ticket_true.i	402	86	21.3930	124130	38599
07_rand_true.i	7318	2504	34.2170	120356	72209
08_rand_cas_true.i	4026	783	19.4486	79716	28258
09_fmaxsym_true.i	679	135	19.8822	76694	27046
10_fmaxsym_cas_true.i	823	776	94.2892	18567	18567
11_fmaxsymopt_true.i	6981	5535	79.2866	106094	100234
12_fmaxsymopt_cas_true.i	2318	1772	76.4452	53810	48948
13_unverif_true.i	1327	27	2.0347	181251	14416
14_spin2003_true.i	1859	47	2.5282	2882659	75961
15_dekker_true.i	255	0	0.0000	273452	442
16_peterson_true.i	4	0	0.0000	9614	53
17_szymanski_true.i	4191	14	0.3340	395961	6015
20_lamport_true.i	2142	1	0.0467	754592	1324

25_stack_true.i	1273	1099	86.3315	105955	98719
26_stack_cas_true.i	3504	3369	96.1473	140503	134069
27_Boop_simple_vf_false.i	9	3	33.3333	8748	2976
28_buggy_simple_loop1_vf_false	1026	1094	106.6277	3817	3817
29_conditionals_vs_true.i	5306	444	8.3679	173027	73947
30_Function_Pointer3_vs_true.i	3302	2704	81.8898	246714	219658
31_simple_loop5_vs_true.i	2255	34	1.5078	2042618	50817
32_pthread5_vs_false.i	8789	12	0.1365	449073	16682
33_double_lock_p1_vs_true.i	1091	1071	98.1668	112830	98073
34_double_lock_p2_vs_true.i	1878	1718	91.4803	132280	119041
35_double_lock_p3_vs_true.i	8854	6465	73.0178	501180	422207
36_stack_cas_p0_vs_concur_true	4603	3663	79.5785	114363	110942
37_stack_lock_p0_vs_concur_tru	5906	5768	97.6634	286164	259645
38_rand_cas_vs_concur_true.i	2974	2886	97.0410	18264	17193
39_rand_lock_p0_vs_true.i	5883	125	2.1248	421378	61330
45_monabsex1_vs_true.i	1674	88	5.2569	856860	77314
47_ticket_lock_hc_backoff_vs_t	1083	329	30.3786	406785	157045
48_ticket_lock_low_contention_	10649	89	0.8358	1355947	71752
dekker_true.i	1991	10	0.5023	2278580	11839
fib_bench_false.i	23	0	0.0000	43830	597
fib_bench_longer_false.i	23	0	0.0000	43830	597
fib_bench_longer_true.i	23	0	0.0000	43830	597
fib_bench_longest_false.i	24	1	4.1667	43830	597
fib_bench_longest_true.i	24	0	0.0000	43830	597
fib_bench_true.i	23	0	0.0000	43830	597
lamport_true.i	718	20	2.7855	499831	13539
lazy01_false.i	932	16	1.7167	412364	14585
peterson_true.i	439	2	0.4556	674597	2690
qrcu_false.i	4966	3584	72.1708	112097	101055
qrcu_true.i	1484	1330	89.6226	112097	101055
queue_false.i	411	49	11.9221	136484	28136
queue_ok_true.i	905	52	5.7459	722966	45814
read_write_lock_false.i	1006	184	18.2903	841504	132079
read_write_lock_true.i	789	148	18.7579	655608	106026
reorder_2_false.i	998	599	60.0200	160660	111301
reorder_5_false.i	4318	808	18.7124	3604813	618407
scull_true.i	643	675	104.9767	93416	88300
sigma_false.i	10965	8823	80.4651	7265574	5450284
sigma_true.i	11372	8794	77.3303	7265574	5450284

singleton_false.i	12562	6449	51.3374	1253886	825713
singleton_true.i	4717	3425	72.6097	1253886	825713
stack_false.i	5243	279	5.3214	357623	94729
stack_true.i	3321	280	8.4312	298289	92010
stateful01_false.i	90	2	2.2222	73675	1676
stateful01_true.i	83	0	0.0000	69241	264
sync01_true.i	118	8	6.7797	71687	3247
szymanski_true.i	2005	53	2.6434	2591205	60366
time_var_mutex_true.i	881	8	0.9081	443694	8044
twostage_3_false.i	566	68	12.0141	400616	52735

TABLE B.3: Performance differences between unoptimised ESBMC and ESBMC with state hashing enabled, ordered by testname. See Appendix B for table format

Test name	Unopt	MPOR	Pct diff	UI	OI
16_peterson_true.i	4	0	0.0000	9614	679
39_rand_lock_p0_vs_true.i	5883	2	0.0340	421378	2897
32_pthread5_vs_false.i	8789	3	0.0341	449073	7901
stack_false.i	5243	4	0.0763	357623	5021
12_fmaxsymopt_cas_true.i	2318	2	0.0863	53810	929
01_inc_true.i	5061	5	0.0988	1286217	11161
queue_ok_true.i	905	1	0.1105	722966	714
time_var_mutex_true.i	881	1	0.1135	443694	959
stack_true.i	3321	4	0.1204	298289	5431
48_ticket_lock_low_contention_	10649	13	0.1221	1355947	22149
11_fmaxsymopt_true.i	6981	9	0.1289	106094	4967
09_fmaxsym_true.i	679	1	0.1473	76694	881
29_conditionals_vs_true.i	5306	9	0.1696	173027	10059
twostage_3_false.i	566	1	0.1767	400616	1382
dekker_true.i	1991	4	0.2009	2278580	7396
queue_false.i	411	1	0.2433	136484	1217
15_dekker_true.i	255	1	0.3922	273452	2390
05_tas_true.i	5529	24	0.4341	558538	42995
13_unverif_true.i	1327	6	0.4521	181251	10402
20_lamport_true.i	2142	10	0.4669	754592	10147
08_rand_cas_true.i	4026	21	0.5216	79716	8152
lazy01_false.i	932	5	0.5365	412364	5873
peterson_true.i	439	3	0.6834	674597	6666
07_rand_true.i	7318	61	0.8336	120356	15890
sync01_true.i	118	1	0.8475	71687	732

stateful01_false.i	90	1	1.1111	73675	1510
stateful01_true.i	83	1	1.2048	69241	1510
02_inc_cas_true.i	637	8	1.2559	112698	9746
szymanski_true.i	2005	28	1.3965	2591205	48202
30_Function_Pointer3_vs_true.i	3302	56	1.6959	246714	27509
14_spin2003_true.i	1859	36	1.9365	2882659	93139
10_fmaxsym_cas_true.i	823	18	2.1871	18567	3385
reorder_5_false.i	4318	98	2.2696	3604813	96446
33_double_lock_p1_vs_true.i	1091	39	3.5747	112830	22486
read_write_lock_true.i	789	29	3.6755	655608	31111
read_write_lock_false.i	1006	38	3.7773	841504	40335
34_double_lock_p2_vs_true.i	1878	71	3.7806	132280	29680
lamport_true.i	718	32	4.4568	499831	22754
26_stack_cas_true.i	3504	183	5.2226	140503	35705
35_double_lock_p3_vs_true.i	8854	507	5.7262	501180	98595
31_simple_loop5_vs_true.i	2255	144	6.3858	2042618	205493
36_stack_cas_p0_vs_concur_true	4603	301	6.5392	114363	38349
25_stack_true.i	1273	84	6.5986	105955	23352
qrcu_false.i	4966	384	7.7326	112097	29672
03_incdec_true.i	1746	140	8.0183	45261	16213
reorder_2_false.i	998	84	8.4168	160660	26730
06_ticket_true.i	402	36	8.9552	124130	27371
scull_true.i	643	68	10.5754	93416	19952
37_stack_lock_p0_vs_concur_tru	5906	668	11.3105	286164	87752
sigma_true.i	11372	1304	11.4668	7265574	1074683
sigma_false.i	10965	1312	11.9653	7265574	1074683
45_monabsex1_vs_true.i	1674	216	12.9032	856860	238604
47_ticket_lock_hc_backoff_vs_t	1083	155	14.3121	406785	104469
qrcu_true.i	1484	255	17.1833	112097	29672
27_Boop_simple_vf_false.i	9	2	22.2222	8748	2101
04_incdec_cas_true.i	1814	427	23.5391	15314	8110
38_rand_cas_vs_concur_true.i	2974	736	24.7478	18264	9334
fib_bench_false.i	23	8	34.7826	43830	20536
fib_bench_longest_true.i	24	9	37.5000	43830	20536
fib_bench_longest_false.i	24	9	37.5000	43830	20536
fib_bench_longer_true.i	23	9	39.1304	43830	20536
fib_bench_true.i	23	9	39.1304	43830	20536
fib_bench_longer_false.i	23	9	39.1304	43830	20536
singleton_false.i	12562	5537	44.0774	1253886	704750

singleton_true.i	4717	2527	53.5722	1253886	728403
28_buggy_simple_loop1_vf_false	1026	644	62.7680	3817	3087
17_szymanski_true.i	4191	2814	67.1439	395961	229512

TABLE B.4: Performance differences between unoptimised ESBMC and ESBMC with the MPOR optimisation enabled, ordered by performance improvement. See Appendix B for table format

Test name	Unopt	MPOR	Pct diff	UI	OI
01_inc_true.i	5061	5	0.0988	1286217	11161
02_inc_cas_true.i	637	8	1.2559	112698	9746
03_incdec_true.i	1746	140	8.0183	45261	16213
04_incdec_cas_true.i	1814	427	23.5391	15314	8110
05_tas_true.i	5529	24	0.4341	558538	42995
06_ticket_true.i	402	36	8.9552	124130	27371
07_rand_true.i	7318	61	0.8336	120356	15890
08_rand_cas_true.i	4026	21	0.5216	79716	8152
09_fmaxsym_true.i	679	1	0.1473	76694	881
10_fmaxsym_cas_true.i	823	18	2.1871	18567	3385
11_fmaxsymopt_true.i	6981	9	0.1289	106094	4967
12_fmaxsymopt_cas_true.i	2318	2	0.0863	53810	929
13_unverif_true.i	1327	6	0.4521	181251	10402
14_spin2003_true.i	1859	36	1.9365	2882659	93139
15_dekker_true.i	255	1	0.3922	273452	2390
16_peterson_true.i	4	0	0.0000	9614	679
17_szymanski_true.i	4191	2814	67.1439	395961	229512
20_lamport_true.i	2142	10	0.4669	754592	10147
25_stack_true.i	1273	84	6.5986	105955	23352
26_stack_cas_true.i	3504	183	5.2226	140503	35705
27_Boop_simple_vf_false.i	9	2	22.2222	8748	2101
28_buggy_simple_loop1_vf_false	1026	644	62.7680	3817	3087
29_conditionals_vs_true.i	5306	9	0.1696	173027	10059
30_Function_Pointer3_vs_true.i	3302	56	1.6959	246714	27509
31_simple_loop5_vs_true.i	2255	144	6.3858	2042618	205493
32_pthread5_vs_false.i	8789	3	0.0341	449073	7901
33_double_lock_p1_vs_true.i	1091	39	3.5747	112830	22486
34_double_lock_p2_vs_true.i	1878	71	3.7806	132280	29680
35_double_lock_p3_vs_true.i	8854	507	5.7262	501180	98595
36_stack_cas_p0_vs_concur_true	4603	301	6.5392	114363	38349
37_stack_lock_p0_vs_concur_tru	5906	668	11.3105	286164	87752

38_rand_cas_vs_concur_true.i	2974	736	24.7478	18264	9334
39_rand_lock_p0_vs_true.i	5883	2	0.0340	421378	2897
45_monabsex1_vs_true.i	1674	216	12.9032	856860	238604
47_ticket_lock_hc_backoff_vs_t	1083	155	14.3121	406785	104469
48_ticket_lock_low_contention_	10649	13	0.1221	1355947	22149
dekker_true.i	1991	4	0.2009	2278580	7396
fib_bench_false.i	23	8	34.7826	43830	20536
fib_bench_longer_false.i	23	9	39.1304	43830	20536
fib_bench_longer_true.i	23	9	39.1304	43830	20536
fib_bench_longest_false.i	24	9	37.5000	43830	20536
fib_bench_longest_true.i	24	9	37.5000	43830	20536
fib_bench_true.i	23	9	39.1304	43830	20536
lamport_true.i	718	32	4.4568	499831	22754
lazy01_false.i	932	5	0.5365	412364	5873
peterson_true.i	439	3	0.6834	674597	6666
qrcu_false.i	4966	384	7.7326	112097	29672
qrcu_true.i	1484	255	17.1833	112097	29672
queue_false.i	411	1	0.2433	136484	1217
queue_ok_true.i	905	1	0.1105	722966	714
read_write_lock_false.i	1006	38	3.7773	841504	40335
read_write_lock_true.i	789	29	3.6755	655608	31111
reorder_2_false.i	998	84	8.4168	160660	26730
reorder_5_false.i	4318	98	2.2696	3604813	96446
scull_true.i	643	68	10.5754	93416	19952
sigma_false.i	10965	1312	11.9653	7265574	1074683
sigma_true.i	11372	1304	11.4668	7265574	1074683
singleton_false.i	12562	5537	44.0774	1253886	704750
singleton_true.i	4717	2527	53.5722	1253886	728403
stack_false.i	5243	4	0.0763	357623	5021
stack_true.i	3321	4	0.1204	298289	5431
stateful01_false.i	90	1	1.1111	73675	1510
stateful01_true.i	83	1	1.2048	69241	1510
sync01_true.i	118	1	0.8475	71687	732
szymanski_true.i	2005	28	1.3965	2591205	48202
time_var_mutex_true.i	881	1	0.1135	443694	959
twostage_3_false.i	566	1	0.1767	400616	1382

TABLE B.5: Performance differences between unoptimised ESBMC and ESBMC with the MPOR optimisation enabled, ordered by testname. See Appendix B for table format

Test name	Unopt	SmtSymex	Pct diff	UI	OI
29_conditionals_vs_true.i	5306	219	4.1274	173027	173027
32_pthread5_vs_false.i	8789	888	10.1035	449073	449073
39_rand_lock_p0_vs_true.i	5883	721	12.2557	421378	421378
38_rand_cas_vs_concur_true.i	2974	386	12.9792	18264	18264
qrcu_false.i	4966	688	13.8542	112097	112097
03_incdec_true.i	1746	255	14.6048	45261	45261
36_stack_cas_p0_vs_concur_true	4603	761	16.5327	114363	114363
stack_false.i	5243	1010	19.2638	357623	357623
04_incdec_cas_true.i	1814	367	20.2315	15314	15314
30_Function_Pointer3_vs_true.i	3302	671	20.3210	246714	246714
34_double_lock_p2_vs_true.i	1878	420	22.3642	132280	132280
stack_true.i	3321	973	29.2984	298289	298289
33_double_lock_p1_vs_true.i	1091	330	30.2475	112830	112830
26_stack_cas_true.i	3504	1172	33.4475	140503	140503
12_fmaxsymopt_cas_true.i	2318	817	35.2459	53810	53810
37_stack_lock_p0_vs_concur_tru	5906	2101	35.5740	286164	286164
11_fmaxsymopt_true.i	6981	2834	40.5959	106094	106094
17_szymanski_true.i	4191	1834	43.7604	395961	395961
qrcu_true.i	1484	671	45.2156	112097	112097
10_fmaxsym_cas_true.i	823	394	47.8736	18567	18567
07_rand_true.i	7318	3600	49.1938	120356	0
09_fmaxsym_true.i	679	337	49.6318	76694	76694
25_stack_true.i	1273	632	49.6465	105955	105955
queue_false.i	411	261	63.5036	136484	136484
twostage_3_false.i	566	477	84.2756	400616	400616
06_ticket_true.i	402	341	84.8259	124130	124130
08_rand_cas_true.i	4026	3600	89.4188	79716	0
lazy01_false.i	932	837	89.8069	412364	412364
47_ticket_lock_hc_backoff_vs_t	1083	991	91.5051	406785	406785
sync01_true.i	118	108	91.5254	71687	71687
stateful01_true.i	83	80	96.3855	69241	69241
13_unverif_true.i	1327	1289	97.1364	181251	181251
16_peterson_true.i	4	4	100.0000	9614	9614
02_inc_cas_true.i	637	645	101.2559	112698	112698
fib_bench_longest_true.i	24	25	104.1667	43830	43830
read_write_lock_false.i	1006	1052	104.5726	841504	841504
read_write_lock_true.i	789	829	105.0697	655608	655608
fib_bench_longest_false.i	24	26	108.3333	43830	43830

lamport_true.i	718	805	112.1170	499831	499831
fib_bench_longer_true.i	23	26	113.0435	43830	43830
fib_bench_false.i	23	26	113.0435	43830	43830
fib_bench_true.i	23	26	113.0435	43830	43830
fib_bench_longer_false.i	23	26	113.0435	43830	43830
scull_true.i	643	765	118.9736	93416	93416
27_Boop_simple_vf_false.i	9	11	122.2222	8748	8748
peterson_true.i	439	539	122.7790	674597	674597
28_buggy_simple_loop1_vf_false	1026	1317	128.3626	3817	3817
stateful01_false.i	90	133	147.7778	73675	73675
reorder_2_false.i	998	1833	183.6673	160660	160660
15_dekker_true.i	255	887	347.8431	273452	273452
singleton_false.i	12562	Crash	N/A	1253886	N/A
35_double_lock_p3_vs_true.i	8854	Crash	N/A	501180	N/A
48_ticket_lock_low_contention_	10649	Crash	N/A	1355947	N/A
sigma_true.i	11372	Crash	N/A	7265574	N/A
sigma_false.i	10965	Crash	N/A	7265574	N/A
01_inc_true.i	5061	Crash	N/A	1286217	N/A
05_tas_true.i	5529	Crash	N/A	558538	N/A
reorder_5_false.i	4318	Crash	N/A	3604813	N/A
dekker_true.i	1991	Crash	N/A	2278580	N/A
szymanski_true.i	2005	Crash	N/A	2591205	N/A
45_monabsex1_vs_true.i	1674	Crash	N/A	856860	N/A
31_simple_loop5_vs_true.i	2255	Crash	N/A	2042618	N/A
20_lamport_true.i	2142	Crash	N/A	754592	N/A
14_spin2003_true.i	1859	Crash	N/A	2882659	N/A
time_var_mutex_true.i	881	Crash	N/A	443694	N/A
queue_ok_true.i	905	Crash	N/A	722966	N/A

TABLE B.6: Performance differences between unoptimised ESBMC and ESBMC with the incremental solving optimisation enabled, ordered by performance improvement. See Appendix B for table format

Test name	Unopt	SmtSymex	Pct diff	UI	OI
01_inc_true.i	5061	Crash	N/A	1286217	N/A
02_inc_cas_true.i	637	645	101.2559	112698	112698
03_incdec_true.i	1746	255	14.6048	45261	45261
04_incdec_cas_true.i	1814	367	20.2315	15314	15314
05_tas_true.i	5529	Crash	N/A	558538	N/A
06_ticket_true.i	402	341	84.8259	124130	124130

07_rand_true.i	7318	3600	49.1938	120356	0
08_rand_cas_true.i	4026	3600	89.4188	79716	0
09_fmaxsym_true.i	679	337	49.6318	76694	76694
10_fmaxsym_cas_true.i	823	394	47.8736	18567	18567
11_fmaxsymopt_true.i	6981	2834	40.5959	106094	106094
12_fmaxsymopt_cas_true.i	2318	817	35.2459	53810	53810
13_unverif_true.i	1327	1289	97.1364	181251	181251
14_spin2003_true.i	1859	Crash	N/A	2882659	N/A
15_dekker_true.i	255	887	347.8431	273452	273452
16_peterson_true.i	4	4	100.0000	9614	9614
17_szymanski_true.i	4191	1834	43.7604	395961	395961
20_lamport_true.i	2142	Crash	N/A	754592	N/A
25_stack_true.i	1273	632	49.6465	105955	105955
26_stack_cas_true.i	3504	1172	33.4475	140503	140503
27_Boop_simple_vf_false.i	9	11	122.2222	8748	8748
28_buggy_simple_loop1_vf_false	1026	1317	128.3626	3817	3817
29_conditionals_vs_true.i	5306	219	4.1274	173027	173027
30_Function_Pointer3_vs_true.i	3302	671	20.3210	246714	246714
31_simple_loop5_vs_true.i	2255	Crash	N/A	2042618	N/A
32_pthread5_vs_false.i	8789	888	10.1035	449073	449073
33_double_lock_p1_vs_true.i	1091	330	30.2475	112830	112830
34_double_lock_p2_vs_true.i	1878	420	22.3642	132280	132280
35_double_lock_p3_vs_true.i	8854	Crash	N/A	501180	N/A
36_stack_cas_p0_vs_concur_true	4603	761	16.5327	114363	114363
37_stack_lock_p0_vs_concur_tru	5906	2101	35.5740	286164	286164
38_rand_cas_vs_concur_true.i	2974	386	12.9792	18264	18264
39_rand_lock_p0_vs_true.i	5883	721	12.2557	421378	421378
45_monabsex1_vs_true.i	1674	Crash	N/A	856860	N/A
47_ticket_lock_hc_backoff_vs_t	1083	991	91.5051	406785	406785
48_ticket_lock_low_contention_	10649	Crash	N/A	1355947	N/A
dekker_true.i	1991	Crash	N/A	2278580	N/A
fib_bench_false.i	23	26	113.0435	43830	43830
fib_bench_longer_false.i	23	26	113.0435	43830	43830
fib_bench_longer_true.i	23	26	113.0435	43830	43830
fib_bench_longest_false.i	24	26	108.3333	43830	43830
fib_bench_longest_true.i	24	25	104.1667	43830	43830
fib_bench_true.i	23	26	113.0435	43830	43830
lamport_true.i	718	805	112.1170	499831	499831
lazy01_false.i	932	837	89.8069	412364	412364

peterson_true.i	439	539	122.7790	674597	674597
qrcu_false.i	4966	688	13.8542	112097	112097
qrcu_true.i	1484	671	45.2156	112097	112097
queue_false.i	411	261	63.5036	136484	136484
queue_ok_true.i	905	Crash	N/A	722966	N/A
read_write_lock_false.i	1006	1052	104.5726	841504	841504
read_write_lock_true.i	789	829	105.0697	655608	655608
reorder_2_false.i	998	1833	183.6673	160660	160660
reorder_5_false.i	4318	Crash	N/A	3604813	N/A
scull_true.i	643	765	118.9736	93416	93416
sigma_false.i	10965	Crash	N/A	7265574	N/A
sigma_true.i	11372	Crash	N/A	7265574	N/A
singleton_false.i	12562	Crash	N/A	1253886	N/A
singleton_true.i	4717	Crash	N/A	1253886	N/A
stack_false.i	5243	1010	19.2638	357623	357623
stack_true.i	3321	973	29.2984	298289	298289
stateful01_false.i	90	133	147.7778	73675	73675
stateful01_true.i	83	80	96.3855	69241	69241
sync01_true.i	118	108	91.5254	71687	71687
szymanski_true.i	2005	Crash	N/A	2591205	N/A
time_var_mutex_true.i	881	Crash	N/A	443694	N/A
twostage_3_false.i	566	477	84.2756	400616	400616

TABLE B.7: Performance differences between unoptimised ESBMC and ESBMC with the incremental solving optimisation enabled, ordered by testname. See Appendix B for table format

Test name	Unopt	TGuard	Pct diff	UI	OI
20_lamport_true.i	2142	34	1.5873	754592	30712
singleton_false.i	12562	332	2.6429	1253886	299175
29_conditionals_vs_true.i	5306	204	3.8447	173027	123874
17_szymanski_true.i	4191	204	4.8676	395961	169131
35_double_lock_p3_vs_true.i	8854	507	5.7262	501180	269577
36_stack_cas_p0_vs_concur_true	4603	284	6.1699	114363	84022
sync01_true.i	118	8	6.7797	71687	11595
singleton_true.i	4717	327	6.9324	1253886	299175
32_pthread5_vs_false.i	8789	627	7.1339	449073	222538
30_Function_Pointer3_vs_true.i	3302	242	7.3289	246714	130590
stack_false.i	5243	396	7.5529	357623	129574
qrcu_false.i	4966	391	7.8735	112097	75131

37_stack_lock_p0_vs_concur_tru	5906	483	8.1781	286164	135324
03_incdec_true.i	1746	151	8.6483	45261	32549
38_rand_cas_vs_concur_true.i	2974	292	9.8184	18264	13372
04_incdec_cas_true.i	1814	182	10.0331	15314	11898
34_double_lock_p2_vs_true.i	1878	206	10.9691	132280	89872
26_stack_cas_true.i	3504	410	11.7009	140503	102406
stack_true.i	3321	465	14.0018	298289	112116
33_double_lock_p1_vs_true.i	1091	166	15.2154	112830	75932
stateful01_true.i	83	13	15.6627	69241	15935
15_dekker_true.i	255	43	16.8627	273452	33881
39_rand_lock_p0_vs_true.i	5883	1047	17.7970	421378	217787
25_stack_true.i	1273	227	17.8319	105955	76705
szymanski_true.i	2005	365	18.2045	2591205	758332
27_Boop_simple_vf_false.i	9	2	22.2222	8748	2691
qrcu_true.i	1484	384	25.8760	112097	75131
queue_false.i	411	109	26.5207	136484	79376
31_simple_loop5_vs_true.i	2255	725	32.1508	2042618	956966
queue_ok_true.i	905	293	32.3757	722966	272383
dekker_true.i	1991	724	36.3636	2278580	811084
time_var_mutex_true.i	881	324	36.7764	443694	186576
read_write_lock_true.i	789	299	37.8961	655608	282901
read_write_lock_false.i	1006	387	38.4692	841504	365129
lazy01_false.i	932	389	41.7382	412364	100798
06_ticket_true.i	402	177	44.0299	124130	79406
peterson_true.i	439	195	44.4191	674597	270893
07_rand_true.i	7318	4321	59.0461	120356	70210
12_fmaxsymopt_cas_true.i	2318	1191	51.3805	53810	21570
14_spin2003_true.i	1859	988	53.1469	2882659	1275675
47_ticket_lock_hc_backoff_vs_t	1083	583	53.8319	406785	298849
lamport_true.i	718	408	56.8245	499831	292908
stateful01_false.i	90	53	58.8889	73675	15935
45_monabsex1_vs_true.i	1674	1045	62.4253	856860	561771
11_fmaxsymopt_true.i	6981	5653	80.9769	106094	73714
13_unverif_true.i	1327	1173	88.3949	181251	121305
scull_true.i	643	584	90.8243	93416	63406
reorder_2_false.i	998	932	93.3868	160660	61497
02_inc_cas_true.i	637	614	96.3893	112698	85688
08_rand_cas_true.i	4026	4032	100.0149	79716	50631
10_fmaxsym_cas_true.i	823	885	107.5334	18567	15192

twostage_3_false.i	566	622	109.8940	400616	117508
28_buggy_simple_loop1_vf_false	1026	1159	112.9630	3817	3537
reorder_5_false.i	4318	7083	164.0342	3604813	1301418
16_peterson_true.i	4	8	200.0000	9614	6868
fib_bench_longest_true.i	24	61	254.1667	43830	31421
fib_bench_longest_false.i	24	61	254.1667	43830	31421
fib_bench_true.i	23	60	260.8696	43830	31421
fib_bench_longer_true.i	23	61	265.2174	43830	31421
fib_bench_false.i	23	61	265.2174	43830	31421
fib_bench_longer_false.i	23	61	265.2174	43830	31421
09_fmaxsym_true.i	679	1841	271.1340	76694	44082
48_ticket_lock_low_contention_	10649	Crash	N/A	1355947	N/A
sigma_true.i	11372	Crash	N/A	7265574	N/A
sigma_false.i	10965	Crash	N/A	7265574	N/A
01_inc_true.i	5061	Crash	N/A	1286217	N/A
05_tas_true.i	5529	Crash	N/A	558538	N/A

TABLE B.8: Performance differences between unoptimised ESBMC and ESBMC with the thread guard optimisation enabled, ordered by performance improvement. See Appendix B for table format

Test name	Unopt	TGuard	Pct diff	UI	OI
01_inc_true.i	5061	Crash	N/A	1286217	N/A
02_inc_cas_true.i	637	614	96.3893	112698	85688
03_incdec_true.i	1746	151	8.6483	45261	32549
04_incdec_cas_true.i	1814	182	10.0331	15314	11898
05_tas_true.i	5529	Crash	N/A	558538	N/A
06_ticket_true.i	402	177	44.0299	124130	79406
07_rand_true.i	7318	4321	59.0461	120356	70210
08_rand_cas_true.i	4026	4032	100.0149	79716	50631
09_fmaxsym_true.i	679	1841	271.1340	76694	44082
10_fmaxsym_cas_true.i	823	885	107.5334	18567	15192
11_fmaxsymopt_true.i	6981	5653	80.9769	106094	73714
12_fmaxsymopt_cas_true.i	2318	1191	51.3805	53810	21570
13_unverif_true.i	1327	1173	88.3949	181251	121305
14_spin2003_true.i	1859	988	53.1469	2882659	1275675
15_dekker_true.i	255	43	16.8627	273452	33881
16_peterson_true.i	4	8	200.0000	9614	6868
17_szymanski_true.i	4191	204	4.8676	395961	169131
20_lamport_true.i	2142	34	1.5873	754592	30712

25_stack_true.i	1273	227	17.8319	105955	76705
26_stack_cas_true.i	3504	410	11.7009	140503	102406
27_Boop_simple_vf_false.i	9	2	22.2222	8748	2691
28_buggy_simple_loop1_vf_false	1026	1159	112.9630	3817	3537
29_conditionals_vs_true.i	5306	204	3.8447	173027	123874
30_Function_Pointer3_vs_true.i	3302	242	7.3289	246714	130590
31_simple_loop5_vs_true.i	2255	725	32.1508	2042618	956966
32_pthread5_vs_false.i	8789	627	7.1339	449073	222538
33_double_lock_p1_vs_true.i	1091	166	15.2154	112830	75932
34_double_lock_p2_vs_true.i	1878	206	10.9691	132280	89872
35_double_lock_p3_vs_true.i	8854	507	5.7262	501180	269577
36_stack_cas_p0_vs_concur_true	4603	284	6.1699	114363	84022
37_stack_lock_p0_vs_concur_tru	5906	483	8.1781	286164	135324
38_rand_cas_vs_concur_true.i	2974	292	9.8184	18264	13372
39_rand_lock_p0_vs_true.i	5883	1047	17.7970	421378	217787
45_monabsex1_vs_true.i	1674	1045	62.4253	856860	561771
47_ticket_lock_hc_backoff_vs_t	1083	583	53.8319	406785	298849
48_ticket_lock_low_contention_	10649	Crash	N/A	1355947	N/A
dekker_true.i	1991	724	36.3636	2278580	811084
fib_bench_false.i	23	61	265.2174	43830	31421
fib_bench_longer_false.i	23	61	265.2174	43830	31421
fib_bench_longer_true.i	23	61	265.2174	43830	31421
fib_bench_longest_false.i	24	61	254.1667	43830	31421
fib_bench_longest_true.i	24	61	254.1667	43830	31421
fib_bench_true.i	23	60	260.8696	43830	31421
lamport_true.i	718	408	56.8245	499831	292908
lazy01_false.i	932	389	41.7382	412364	100798
peterson_true.i	439	195	44.4191	674597	270893
qrcu_false.i	4966	391	7.8735	112097	75131
qrcu_true.i	1484	384	25.8760	112097	75131
queue_false.i	411	109	26.5207	136484	79376
queue_ok_true.i	905	293	32.3757	722966	272383
read_write_lock_false.i	1006	387	38.4692	841504	365129
read_write_lock_true.i	789	299	37.8961	655608	282901
reorder_2_false.i	998	932	93.3868	160660	61497
reorder_5_false.i	4318	7083	164.0342	3604813	1301418
scull_true.i	643	584	90.8243	93416	63406
sigma_false.i	10965	Crash	N/A	7265574	N/A
sigma_true.i	11372	Crash	N/A	7265574	N/A

singleton_false.i	12562	332	2.6429	1253886	299175
singleton_true.i	4717	327	6.9324	1253886	299175
stack_false.i	5243	396	7.5529	357623	129574
stack_true.i	3321	465	14.0018	298289	112116
stateful01_false.i	90	53	58.8889	73675	15935
stateful01_true.i	83	13	15.6627	69241	15935
sync01_true.i	118	8	6.7797	71687	11595
szymanski_true.i	2005	365	18.2045	2591205	758332
time_var_mutex_true.i	881	324	36.7764	443694	186576
twostage_3_false.i	566	622	109.8940	400616	117508

TABLE B.9: Performance differences between unoptimised ESBMC and ESBMC with the thread guard optimisation enabled, ordered by testname. See Appendix B for table format

Change in time	Test name
-7901	32_pthread5_vs_false.i
-5162	39_rand_lock_p0_vs_true.i
-5087	29_conditionals_vs_true.i
-4278	qrcu_false.i
-4233	stack_false.i
-4147	11_fmaxsymopt_true.i
-3842	36_stack_cas_p0_vs_concur_true.i
-3805	37_stack_lock_p0_vs_concur_true.i
-3733	37_stack_lock_p0_vs_concur_true.i
-3675	35_double_lock_p3_vs_true.i
-3667	stack_false.i
-3482	32_pthread5_vs_false.i
-2631	30_Function_Pointer3_vs_true.i
-2588	38_rand_cas_vs_concur_true.i
-2527	30_Function_Pointer3_vs_true.i
-2468	stack_false.i
-2357	17_szymanski_true.i
-2348	stack_true.i
-2332	26_stack_cas_true.i
-2305	01_inc_true.i
-2239	29_conditionals_vs_true.i
-2158	stack_true.i
-2111	11_fmaxsymopt_true.i
-1700	singleton_false.i
-1653	singleton_false.i

-1603	singleton_false.i
-1526	singleton_false.i
-1523	singleton_false.i
-1501	12_fmaxsymopt_cas_true.i
-1491	03_incdec_true.i
-1479	stack_true.i
-1458	34_double_lock_p2_vs_true.i
-1447	04_incdec_cas_true.i
-1364	qrcu_false.i
-1327	17_szymanski_true.i
1190	07_rand_true.i
-1161	38_rand_cas_vs_concur_true.i
-1095	29_conditionals_vs_true.i
992	reorder_2_false.i
908	reorder_2_false.i

TABLE B.10: List of most significant changes in performance between an unoptimised version of ESBMC and one with incremental solving, across all runs.

Appendix C

ESBMCs SMT encoding

Before examining ESBMC's SMT encoding, it is important to first recognise that ESBMC inherits its verification approach and a large amount of its code base from the CBMC model checker [53]. The methods described in this section are thus not novel contributions by myself, although some of them may not have appeared in publications before. It is also important that ESBMC was branched from CBMC at version 2.9, and in the intervening six years we have diverged significantly. The most notable difference is that CBMC focuses on reducing program traces to SAT[1], while ESBMC focuses on SMT solvers. From here on in this section I will only mention ESBMCs encoding to SMT.

I perform this description in two sections. In the first, some of the higher level aspects of the environment that the program executes in are discussed, necessary as the program is not executing over a real-world machine or even simulation of one. Secondly, specific expressions and features of the C language and how they are encoded to SMT are discussed.

One property of nomenclature occurs: an "assertion" in C is a condition representing a property of the program that must never be violated. In SMT, however, an assertion is a constraint on the valuation of the formula that must be met in order for the formula to be satisfiable. In this chapter I refer to C language assertions as "property assertion"s.

C.1 Executing C as a nondeterministic program

The C language specification itself [7] defines C as operating over an *abstract machine*. Abstract in that the standard does not specify many of the features of the real machines (such as the size of pointers), but a machine in that the program is expected to posses a state and for program instructions to modify that state. It's initial state, and the environment in which it operates, are also defined by the C standard.

[1] In recent years I understand some SMT solvers have gained some support in CBMC, but are still considered experimental

Model checking diverges from this model of operation because it attempts to check the correctness of program properties for all possible inputs. Such inputs are *nondeterministic*, in that their value can be anything (within the bounds of the C type system). As a result, any model checker must explore multiple machine states to verify properties. In symbolic bounded model checking, this is done by translating the program into first order logic formula that represents all possible states in the program, modulo the program unwinding bound. In first order logic, however, every variable must be explicitly identified, is immutable, and no indirection is permitted, presenting several encoding challenges.

C.1.1 Renaming variables

The first step to encoding programs to SMT, is to allow nondeterministic values to be free variables in the formula. Using an unconstrained variable will cause the solver to consider every single valuation of it, and whether it leads to the formula being satisfied (and consequently, a property being violated). Assignments to variables are produced by creating an SMT expression of the right hand side of the assignment, creating an SMT variable for the left hand side, and encoding an SMT assertion that one equals the other. The form of SMT expressions is discussed below. To work around the immutability of SMT variables, a policy of *renaming* is pursued: once an assignment is made to a variable, subsequent assignments to the same variable are given a slightly different name. Consider a loop that increments an iterator `i` three times. The result is *four* SMT variables: one for the initial state of the variable, and three for the assignments. To illustrate, the constraints in the SMT formula are thus:

```
i#1 == i#0 + 1
i#2 == i#1 + 1
i#3 == i#2 + 1
```

The left hand side variables are numbered one to three, each one representing the value of an individual assignment. Note that the right hand side uses the variable name from before the current assignment: this corresponds to the right hand side of an expression in C being evaluated before the left hand side is assigned. The effect of this is that the model checker must always keep track of what the most recent assignment to a lexical variable is, while symbolically executing a program.

A lexical variable does not only have multiple values: it can also exist in multiple different contexts, most obviously in a recursive function. In C, this is referred as *storage*: whenever a new function (or thread) scope is created during execution, memory is allocated to store the variable.[2] On a machine, assignments to such variables know which storage is in scope, and write to the relevant piece of memory. In ESBMC, this is replicated by introducing an additional level of renaming, to identify the piece of storage that is being assigned to. Thus, the assignments

[2] In the C spec, these are referred to as *automatic storage* and *thread local storage* duration, respectively

```
int a;

if (nondet_bool()) {
        a = 0;
} else {
        a = 1;
}

assert (a == 0);
```

FIGURE C.1: A piece of code with a nondeterministic control flow graph

above would be named `i@1#1` or similar, and lexical variables with different storage would have different numbers after the "@" character. Distinguishing storage in different threads is achieved in exactly the same manner. This requires the model checker to track what stack frame and what thread are currently being executed, to decide which piece of storage to assign (or read).

Within ESBMC, the variable assignment level of renaming is referred to as "level 1", and storage as "level 2"; I will use these terms from here.

C.1.2 Path exploration and guards

As well as having to resolve which variable valuation to use when symbolically executing a piece of code, complexity can also be caused by the control flow graph (CFG) of the program. Consider Figure C.1, where control flow branches depending on a nondeterministic decision, merges, and the subsequent property assertion depends on which path was taken. One solution to this would be to explicitly enumerate every path through the program and verify each individually. This, however, is needlessly explicit, as we can achieve the same in a single formula. Upon reaching a CFG branch, ESBMC executes both paths from the branch, until they merge or the current function runs. Renaming records are duplicated such that both paths can read variables from their common history, but new assignments (and new level 2 storage) in each path have disjunct names. Finally, when control flow merges, all the level 2 variables[3] that have been assigned to in either path are *merged*, through the use of the SMT *phi* function.

The merge procedure is to take each modified variable and find the most recent assignment to it on each of the merged paths.[4] Then, a new assignment is made with the *phi* or "if-then-else" function on the right hand side, which selects between the value from one path of the program or the other, depending on the branch condition. Following the example in Figure C.1, the assignments that it would produce are below:

```
a@1#1 == 0
a@1#2 == 1
a@1#3 == (nondet_val_1) ? a@1#1 : a@1#2
```

[3]i.e., all variables with storage

[4]This may be an assignment from before the control flow branch on one branch, or if the variable was uninitialised and not assigned, a free variable is used

As the `if` statement does not make an assignment, it does not have any representation in the SMT formula. Regardless of the truth of the branch condition, the assignments down each path (to `a@1#1` and `a@1#2`) are performed. The assignment to `a@1#3` represents the merge of the two branches at the end of the `if` statement, with the *phi* function shown as a C tertiary operator. If the condition in the `if` statement is true, then `a@1#1` is assigned to the new value of `a`; however if it is false, then `a@1#2` is assigned to `a`. Following the merge, new uses of the merged variables refer to the `phi` assignment of the variable. In our example, the `assert` call at the end of Figure C.1 checks the value of `a@1#3`, and the solver is then free to explore the possible values of `a` from both paths of the branch.

This approach is not limited to simple fork-and-join branches: loops that contain a conditional `break` can result in several branches that must all be merged together once exploration of the loop halts. In these cases, pairs of paths are merged together into one until only one remains. The order is unimportant. It is also possible for additional branches to occur down one path, and for the paths to not merge symmetrically (i.e., merging in the order that they were branched). This makes it important for each path to have an associated *guard*, a predicate that holds in all variable valuations where the current path will be taken by the program under test. It is a conjunction of all the branch conditions that must hold to reach the current path. The guard of one path is used as the condition for *phi* functions when merging paths.

The guard is also used to guard any property assertions made on a particular path. The property assertion should only lead to a verification failure if the condition is false and the path to it is a legitimate path through the program.

I omit a description of how ESBMC performs the actual exploration of the control flow graph as it is irrelevant to the SMT encoding. Suffice to say, it produces a set of paths through a function, and the assignments and property assertions that are found along those paths. Function calls are handled as the creation of a new stack frame and associated renaming numbers for variables local to that stack frame, assignment of arguments to parameters, then exploration of paths through that function. Returning is handled by performing an unconditional branch to the end of the function, where all paths are merged, control is returned to the call site and the left stack frame erased. Return values are encoding by encoding an assignment to a special variable when interpreting a return statement: it is then merged with other paths as any other variable would be, before being assigned to the return value at the call site as the function exit occurs.

C.1.3 Bounding paths

One of the more significant mechanisms in the CFG exploration procedure is how loops are bounded. ESBMC defines a loop as any backwards branch in a function, so the use of explicit `goto` instructions are recognised as loops too. As ESBMC is a bounded model checker, we do not explore a backwards branch once it has been followed as many times as the bound allows, and execution continues as if the branch had not been taken. If it was an unconditional backwards branch then there is no next state, and ESBMC finds the closest unmerged path to the backwards branch (typically an exit branch from the loop) and continues exploration from there.

This leaves the problem of what happens to the path truncated by the unwind bound. If the backwards branch is unconditional then the path simply ceases to exist, but if it is conditional then exploration continues, no matter what the condition evaluated to. This can lead to the inaccurate scenario where an expression effectively evaluates to false in a branch condition, then true in a subsequent context. To avoid this, an *unwinding assumption* is encoded, as an SMT assertion (i.e., constraint), that the backwards branch condition always evaluates to false. This causes the SMT solver to discard any program state where the backwards loop would have been followed, as it would be inconsistent with the unwinding assumption. In effect, only paths that loop fewer times than the unwind bound are considered by the solver. A flaw in this approach is that, if a loop with a fixed number of iterations and no other exit branch is bounded before it completes looping, then all code after the loop becomes unreachable as the only path through the program loops more times than the unwind bound. The unwinding assumptions can be disabled with the `--partial-loops` option, however this can lead to invalid states, as discussed. A similar arrangement is used to bound the number of times that recursive functions can recurse.

Another feature (inherited from CBMC) is that of *unwinding assertions*, which instead of assuming that there are no paths through the program that loop more than the unwind bound, asserts that fact instead. This triggers a property assertion failure if such a looping path can be found through the program under test. While this is not necessarily useful for verification, it allows an engineer to discover whether their unwinding bound is causing paths to be discarded. While this is of course the whole point of bounded model checking, in some circumstances it is useful to know for a program with finite state and duration that the model checker has fully explored it.

C.1.4 Dynamic memory allocation

The two mechanisms via which C supports infinite state is through infinite recursion, and dynamically allocated memory. Infinite recursion is necessarily bounded (see above) for verification to complete; however the program under test is able to dynamically allocate memory by using the `malloc` function and others in its family. ESBMC supports this by explicitly identifying each dynamically allocated “hunk” of memory at the point where it is allocated, and adding new variables to the program to hold its value. In essence, memory is never actually allocated, instead we retrospectively add variables to the program as required.

C.2 Translating C expressions to SMT

Having explored how ESBMC models C's abstract state machine in an SMT formula, I now consider how individual C expressions are encoded to SMT. Immediately, it is convenient that SMT supports variables with sorts[5] such as integers, bitvectors, reals, and arrays. ESBMC can encode C integers as SMT integers or bitvectors, however using SMT integers can lead to inaccuracies as they do not have a byte representation or individual bits to manipulate. The rest of this section will only deal with ESBMC's encoding to bitvectors and arrays. This choice introduces inaccuracy elsewhere, as ESBMC must floating point numbers as bitvectors. The overhead of modelling IEEE754 operations is enormous, and so fixed point arithmetic is used instead. While less accurate, the embedded and low level systems that ESBMC is typically applied to tend not to use floating point numbers as there is little call for such features at that level, and limited hardware support.

Each C variable is thus modelled in SMT as a bitvector or array (structs, unions and pointers are discussed below). The majority of C operators are already available for use in the SMTLIB standard, including all arithmetic and bit operations. C expressions can thus be directly translated from C variables, constants and operators, to SMT variables, constants and operators. Likewise, array selection and storage is directly supported by the solver.

C.2.1 Assertions and assumptions

Property assertions encountered during the symbolic execution of the program under test are conditions that must always hold, and as a result the SMT solver must search for any consistent variable valuation of the program that violates a property assertion. Conversely, one may also encode *assumptions* in the program, that place a constraint on the states explored. The use case of assumptions is that a verification engineer may wish to constraint the operation of the program in some symbolic way, for example testing a path only with a certain variable valued between one and one hundred.

As a model checker, we require that any property violation must have the program trace available as a counterexample to program correctness, for the verification engineer to examine. When instructing the SMT solver to search for property violations, the solver must find a satisfying assignment when a property is violated, so that we can examine the variable assignments. To achieve this, after all assignments are encoded to the formula, a final constraint is added stating that the formula is consistent only when *at least one* property assertion is violated. Multiple property assertion violations are entirely possible, and in fact likely given that after the first one, the program is known to be incorrect. This constraint is formulated by taking every property assertion condition in the program, inverting it (so that it evaluates to true if the property is violated), then asserting that the disjunction of all the conditions evaluates to true.

[5]The SMTLIB specification defines *sorts* of logical formula, where programmers may think of types

```
int *p, a = 0, b = 1;

if (nondet_bool()) {
        p = &a;
} else {
        p = &b;
}

assert(*p);
```

FIGURE C.2: A piece of code with a nondeterministic pointer dereference

Assumptions are encoded by taking the conjunction of all assumption conditions, and guarding the property assertion condition. The result is that the formula can only be satisfiable if all the assumptions applied to the program evaluate to true.

C.2.2 Indirection

The first serious hurdle is dealing with the indirection that is permitted by the C specification. Pointers indirectly refer to a particular `data object`, i.e. the storage for a variable, and access to it occurs through dereferencing the pointer and reading or writing the corresponding variable. In symbolic model checking, at a particular point in a path, a pointer may potentially point at one of a set of data objects, depending on past conditions. This cannot be encoded to SMT, as there is no facility for indirection.

Instead, ESBMC tracks the set of data objects that a pointer points at, and upon dereferencing the pointer, produces an expression that selects which data object to evaluate to through the use of *phi* operations. If the pointer dereference is assigned to, this becomes a set of conditional assignments to data objects.

The problem is now reduced from full indirection, down to the ability to identify which data object a pointer points at, in the SMT solver. To implement this, ESBMC (initially) models pointer variables as an integer, and gives every data object in the program trace a unique number. Then, identifying the data object in the SMT solver is a case of enumerating which data objects it *might* be, and comparing their data object numbers to the pointer variable value. Consider Figure C.2, where at the final statement `p` may point at either `a` or `b`. Down each branch of the condition, `p` will be assigned the data object number of `a` or `b`, and when the two paths merge, the `p` variable will be merged in a *phi* function just like any other variable. ESBMC will also statically track the set of data objects `p` may point at, and merge that set when two paths merge. For the final dereference, an expression similar to this will be produced:

((p == 1) ? a : b)

Assuming that the data object number of p is one.

This does not fully model the potential behaviours that pointers can exhibit. Pointer arithmetic can allow pointers to point *inside* a data object, rather than at the start of it. This could be a character pointer to a byte inside an integer, or an integer pointer to an element of an array. The *offset* into the data object may not be statically determinable, and so to model this in the SMT solver, pointer variables become a pair of variables,[6] one containing the data object number, and the other containing the offset into the object. This allows the solver to identify a data object and how far into it the pointer points; constructing an expression to accurately represent this such a reference is difficult, and discussed in more detail in Section 4.2.

Numerous things can go wrong when dereferencing a pointer: the pointer may be a NULL pointer, it may point outside the bounds of the data object, or not point at any data object at all (if the program casts an integer to a pointer, for example). Happily, all these circumstances are defined by the C standard to be undefined behaviour, and thus a program error. In these circumstances, ESBMC encodes a property assertion during symbolic execution that there is no valid path to such a state, and to make the SMT formula well formed, inserts a free variable to be read or written. In any case where an SMT expression does evaluate to such a free variable, the property assertion should trigger a verification failure.

C.2.3 Address space

The C language allows for pointers to be freely cast to and from integers of a sufficient bit width. While the comparison of pointer variables that do not point at the same data object is undefined behaviour, it is legal for a program to make decisions based on the bit pattern representation of a pointer. This is usually unwise, although legitimate uses can be contrived, such as using a pointer when computing the hash value of a data object when placing it in a hash table. This means that the problem space of the model checker extends to deciding *where* in memory a data object is located, and whether a program violates a property if data objects are arranged in a certain way.

The C specification itself gives few guarantees about the bit representation of pointers. The null pointer constant must evaluate to zero when cast to an integer, and any identical pointers cast to integers must compare the same. Matters such as comparing the address of later bytes in data structures greater than earlier bytes are not required for the bit representation. This means there is great scope for different behaviours between machines within the C standard, much of which is difficult to symbolically implement.

ESBMC takes a pragmatic approach and follows the memory address space found in most machines, where memory is an array of bytes indexed by an integer, and data objects are placed in this array, with aggregate / composite data structures having members arranged in order of their declaration. Padding withing the address space is present, however it is a fixed amount appropriate for the machine mode that ESBMC is operating in (32 or 64 bits), and no attempt is made to represent all the padding configurations that machines are allowed to use. The location of data objects in memory, however, is encoded symbolically. The start and end address of each

[6] Actually a tuple, see below

data objects are made free integer variables, and all casts to and from pointers and integers are made relative to those variables. Constraints are then encoded that the address range of one object does not overlap the address range of any other object (and that it is sufficiently aligned for the allocated data object). The no-overlap constraint must be applied to each pairing of data objects, making it $O(n^2)/2$ in complexity. The solver is then free to re-arrange the address space location of any data object in a consistent manner, to search for any orderings that may affect the program state.

C.2.4 Casts

The SMTLIB standard prescribes a strong type system in SMT formula: all operations must have exactly matching operand sorts, and no coercion of values is permitted. This means that any cast encountered in the C program must be explicitly encoded as bit operations on the underlying variables. C's type system is weak, so there are many conversions to consider.

Numerous casts do not require a conversion however: the SMTLIB type system does not assign sign or unsigned attributes to bitvectors, signedness is instead encoded in the operations applied to variables. Pointers, represented as as a pair of variables (see Section C.2.2), do not require any conversion either.

Within the domain of integers, when a typecast is encountered the corresponding C rules are applied to the underlying SMT bitvector. Casting a smaller integer to a large one results in an integer promotion. If the C language variable is unsigned, the integer is concattenated with an appropriate number of zeros, and if it is signed then the topmost bit is replicated an appropriate number of times and concatenated with the variable. Likewise, a conversion to a smaller sized integer will result in an SMT extraction being encoded, to extract the appropriate number of lower bits from the variable.

The fixed point approximate representation of floating point numbers also follows the obvious approach: casts from floats to integers extract only the integer part of the underlying bitvector, and casting from integer to float concatenates an appropriate number of zeros representing the fractional part of the float.

The C99 standard introduced boolean types, which are also supported as a native sort by the SMTLIB standard. Casting any integer to a boolean is converted to a comparison of the integer with zero, the result of which is inverted. Casting from a boolean to integer compares the boolean with true, evaluating to one if it was true, and zero otherwise.

More pronounced complexity occurs when casting to and from the pointer representation to an integer. Casting to an integer is straightforwards: an (SMT) array is maintained of the start addresses of each data object, and upon conversion this array is indexed with the data object number from the pointer variable, and then added to the pointer offset in the pointer variable. Producing an array to map in the opposite direction, however, from all possible addresses to their corresponding data object number and offset, is an unfeasible task. Instead, such a cast becomes a chain of comparisons, testing whether the source integer lies in the address space range of a particular data object, and evaluating to that data object number if it does. This scales linearly with the number of data objects (that have their address taken) in the program. If the address cast to a pointer does not lie in the range of any data object, then the data object number is set to a special "invalid" number, which dereference property assertions check for before dereferencing the pointer.

C.2.5 Structs and Unions

Structs and unions are fundamental parts of the C programming language, however there is no analogous sort in the SMTLIB standard. Some SMT solvers (such as Z3) provide their own support for structures, in the form of *tuples*, which are groupings of other variables. Member variables are ordered, and can be explicitly projected or updated. Tuples can also be used in *phi* functions and equalities, allowing their grouped values to be part of variable assignments, like any other variable in a program. Previously, ESBMC has relied upon Z3's tuple implementation rather than reducing structures to an encoding recognised by the SMTLIB standard. Part of my work has been to improve this situation, and is covered in detail in Section 4.1.

Unions are not fully supported by ESBMC. The most difficult feature to implement is that of shared-storage variables, i.e. the fact that the same piece of memory may be written as an integer in one field of a union, then read as a pointer in another field. Implementing this would require every read or write to the union to be decomposed into a series of casts. Rather than do this, the current solution is to represent the union as a tuple, with each field in the tuple representing a field in the union. No shared-storage is performed. This does not meet the C specifications requirements, but does work effectively in the vast majority of use cases, where unions are used to merge several mutually exclusive data records to save memory.

I speculate that the most effective solution to this problem would be to statically determine when unions are used in such a way that casts would be required, and model them in SMT as arrays of bytes rather than any other data structure. For circumstances where only one field of the union is ever used, the current approach is sufficient and accurate.

References

[1] Coverity code adviser. http://www.coverity.com/products/code-advisor/.

[2] Cvc4: the smt solver. http://cvc4.cs.nyu.edu/web/.

[3] Heartbleed vulnerability cve. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[4] How SQLite is Tested. http://www.sqlite.org/testing.html.

[5] Pvs-studio. http://www.viva64.com/en/pvs-studio/.

[6] Sv-comp svn repository. https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c.

[7] ISO/IEC 9899:2011 Information technology — Programming languages — C. December 2011.

[8] Federal Information Processing Standard 180-2. *Secure Hash Standard*. National Institute of Standards and Technology, 2002.

[9] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 49 of *POPL '14*, pages 373–384, New York, NY, USA, 2014. ACM.

[10] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In *ACM Symposium on Principles of Programming Languages, POPL '14*, pages 373–384. ACM, 2014.

[11] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 141–157, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 141–157, Berlin, Heidelberg, 2013. Springer-Verlag.

[13] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[14] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[15] Atmel. Atmel studio 6, for avr and arm. http://www.atmel.com/microsite/atmel_studio6/.

[16] Franz Baader, Andreas Bauer, and Marcel Lippmann. Runtime verification using a temporal description logic. In *Proceedings of the 7th International Conference on Frontiers of Combining Systems*, FroCoS'09, pages 149–164, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] T Ball and S.K. Rajamani. Slic: A specification language for interface checking. *Technical Report MSR-TR-2001-21*, 2002.

[18] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In Michael Burke and Mary Lou Soffa, editors, *PLDI*, pages 203–213. ACM, 2001.

[19] J. Barnat, L. Brim, and P. Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods Symposium*, volume 7226 of *LNCS*, pages 252–267. Springer, 2012.

[20] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[21] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. Divine 3.0 – an explicit-state model checker for multithreaded c & c++ programs. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer Berlin Heidelberg, 2013.

[22] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.

[23] Clark Barrett, Morgan Deters, Leonardo Moura, Albert Oliveras, and Aaron Stump. 6 years of smt-comp. *J. Autom. Reason.*, 50(3):243–277, March 2013.

[24] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[25] Andreas Bauer and Patrik Haslum. LTL goal specifications revisited. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 881–886. IOS Press, 2010.

[26] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007.

[27] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[28] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[29] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of rctl formulas. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 184–194, London, UK, UK, 1998. Springer-Verlag.

[30] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[31] Dirk Beyer. Competition on software verification - (sv-comp). In *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 504–524. Springer, 2012.

[32] Dirk Beyer. Second competition on software verification - (summary of sv-comp 2013). In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 594–609. Springer, 2013.

[33] Dirk Beyer. Status report on software verification. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 373–388. Springer Berlin Heidelberg, 2014.

[34] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[35] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, October 2007.

[36] Dirk Beyer, ThomasA. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer Berlin Heidelberg, 2007.

[37] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[38] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.

[39] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial strength distributed explicit state model checking. In *Proceedings of the 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, PDMC-HIBI '10, pages 28–36, Washington, DC, USA, 2010. IEEE Computer Society.

[40] Johannes Borgström, Shuqin Huang, Magnus Johansson, Palle Raabjerg, Björn Victor, Johannes Åman Pohjola, and Joachim Parrow. Broadcast psi-calculi with an application to wireless protocols. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, volume 7041 of *Lecture Notes in Computer Science*, pages 74–89. Springer Berlin Heidelberg, 2011.

[41] Dragan Bošnački, Leszek Holenderski, and Dennis Dams. A heuristic for symmetry reductions with scalarsets. In JoséNuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 518–533. Springer Berlin Heidelberg, 2001.

[42] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[43] Frederick P. Frederick Phillips Brooks. *The mythical man-month : essays on software engineering*. Reading, Mass. Addison-Wesley Pub. Co., 1975.

[44] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[45] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[46] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[47] J. Richard Büchi. Symposium on decision problems: On a decision method in restricted second order arithmetic. In Patrick Suppes Ernest Nagel and Alfred Tarski, editors, *Logic, Methodology and Philosophy of Science Proceeding of the 1960 International Congress*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 11. Elsevier, 1966.

[48] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[49] Andy Chou. On Detecting Heartbleed with Static Analysis. http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html.

[50] Chun-Nan Chou, Chen-Kai Chu, and Chung-Yang (Ric) Huang. Conquering the scheduling alternative explosion problem of systemc symbolic simulation. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 685–690, Piscataway, NJ, USA, 2013. IEEE Press.

[51] Alessandro Cimatti, Alberto Griggio, BastiaanJoost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and ScottA. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2013.

[52] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[53] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[54] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ansi-c programs using sat. In *Formal Methods in System Design (FMSD), 25:105– 127, September–November*, page 2004, 2003.

[55] Edmund M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[56] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[57] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 450–462, London, UK, UK, 1993. Springer-Verlag.

[58] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. *Form. Methods Syst. Des.*, 10(1):47–71, 1997.

[59] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[60] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96. Springer, 2004.

[61] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[62] Edmund M. Clarke and Flavio Lerda. Model checking: Software and beyond. *J. UCS*, 13(5):639–649, 2007.

[63] SMTLIB Competition Committee. Summary of results for smt-comp 2011. http://smtexec.cs.uiowa.edu/exec/?jobs=856.

[64] SMTLIB Competition Committee. Summary of results for smt-comp 2012. http://smtexec.cs.uiowa.edu/exec/?jobs=856.

[65] SMTLIB Competition Committee. Summary of results for smt-comp 2014. http://smtcomp.sourceforge.net/2014/results-summary.shtml?v=1403902163.

[66] Lucas Cordeiro. *SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems*. PhD thesis, University of Southampton, April 2011.

[67] Lucas Cordeiro, Raimundo S. Barreto, Rafael Barcelos, Meuse N. Oliveira Jr., Vicente Lucena, and Paulo Romero Martins Maciel. Agile development methodology for embedded systems: A platform-based design approach. In *ECBS*, pages 195–202. IEEE Computer Society, 2007.

[68] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 331–340. ACM, 2011.

[69] Lucas Cordeiro, Bernd Fischer, Huan Chen, and João Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In Tianzhou Chen, Dimitrios N. Serpanos, and Walid Taha, editors, *ICESS*, pages 396–403. IEEE, 2009.

[70] Lucas Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *ASE*, pages 137–148. IEEE Computer Society, 2009.

[71] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[72] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[73] SV-COMP chair Dirk Beyer. Sv-comp benchmark repository. `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c`.

[74] Alastair F. Donaldson. *Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking*. Phd thesis, Department of Computing Science University of Glasgow, 2007.

[75] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[76] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[77] B. Dutertre and L. de Moura. The Yices SMT solver. August 2006.

[78] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In Warren A. Hunt and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.

[79] E.Allen Emerson and Thomas Wahl. Dynamic symmetry reduction. In Nicolas Halbwachs and LenoreD. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 382–396. Springer Berlin Heidelberg, 2005.

[80] José Miguel Faria, João Martins, and Jorge Sousa Pinto. An approach to model checking ada programs. In *Proceedings of the 17th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'12, pages 105–118, Berlin, Heidelberg, 2012. Springer-Verlag.

[81] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[82] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[83] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *ASE*, pages 412–416. IEEE Computer Society, 2001.

[84] Patrice Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, 1995.

[85] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[86] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 438–449, London, UK, UK, 1993. Springer-Verlag.

[87] Systems Verification Group. Systems verification group homepage. `http://www.cprover.org/`.

[88] Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 40–47, New York, NY, USA, 2014. ACM.

[89] Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 40–47, New York, NY, USA, 2014. ACM.

[90] Anping He, Jinzhao Wu, and Lian Li. An efficient algorithm for transforming LTL formula to Büchi automaton. In *Proceedings of the 2008 International Conference on Intelligent Computation Technology and Automation - Volume 01*, ICICTA '08, pages 1215–1219, Washington, DC, USA, 2008. IEEE Computer Society.

[91] Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems: First International*

Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.

[92] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[93] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[94] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Softw. Test., Verif. Reliab.*, 11(2):65–79, 2001.

[95] GerardJ. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

[96] Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. The rers grey-box challenge 2012: Analysis of event-condition-action systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 608–614. Springer Berlin Heidelberg, 2012.

[97] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[98] IAR. C/C++ Compiler ARM and Cortex. `http://www.iar.com/Products/IAR-Embedded-Workbench/ARM/`.

[99] Texas Instruments. Code composer studio ide for msp430 microcontrollers. `http://www.ti.com/tool/ccstudio-msp430`.

[100] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded c programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer International Publishing, 2014.

[101] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded c programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer International Publishing, 2014.

[102] ISO/IEC. *Information technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, ISO/IEC/IEEE 9945:2009*. ISO, Geneva, Switzerland, 2009.

[103] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, 2009.

[104] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theor. Comput. Sci.*, 167(1&2):47–72, 1996.

[105] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Lct: A parallel distributed testing tool for multithreaded java programs. *Electron. Notes Theor. Comput. Sci.*, 296:253–259, August 2013.

[106] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 398–413, Berlin, Heidelberg, 2009. Springer-Verlag.

[107] Hans W. Kamp. *Tense Logic and the Theory of Linear Order*. Phd thesis, Computer Science Department, University of California at Los Angeles, USA, 1968.

[108] Andrey Karpov. A Boring Article About a Check of the OpenSSL Project [sic]. http://www.viva64.com/en/b/0250/.

[109] kernel.org. Linux device drivers. https://www.kernel.org/doc/htmldocs/device-drivers/.

[110] Saul A. Kripke. Semantical analysis of modal logic i. normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[111] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[112] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[113] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 509–524. Springer, 2009.

[114] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[115] Leslie Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.

[116] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[117] MIRA Limited. *Guidlines for the use of the C language in critical systems*. MIRA Limited, Watling Street, Nuneaton, UK, 2004.

[118] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[119] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[120] Kenneth L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 250–264, London, UK, UK, 2002. Springer-Verlag.

[121] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: bounded model checking of c and c++ programs using a compiler ir. In *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments*, VSTTE'12, pages 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[122] Microchip. Mplab xc compilers. http://www.microchip.com/pagehandler/en_us/devtools/mplabxc/.

[123] Microsoft. Kernel mode driver reference. http://msdn.microsoft.com/en-us/library/windows/hardware/ff553217(v=vs.85).aspx.

[124] Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006.

[125] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Context-bounded model checking of ltl properties for ansi-c software. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 302–317, Berlin, Heidelberg, 2011. Springer-Verlag.

[126] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Handling unbounded loops with esbmc 1.20. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 619–622, Berlin, Heidelberg, 2013. Springer-Verlag.

[127] Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Applying symbolic bounded model checking to the 2012 RERS greybox challenge. *STTT*, 16(5):519–529, 2014.

[128] Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Model checking LTL properties over ANSI-C programs with bounded traces. *Software and System Modeling*, 14(1):65–81, 2015.

[129] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002.

[130] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455. ACM, 2007.

[131] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[132] Anh Cuong Nguyen and Siau Cheng Khoo. Towards automation of LTL verification for java pathfinder. In *Proceedings of the 15th National Undergraduate Research Opportunities Programme Congress, Singapore*, 2010.

[133] Eric Noonan, Eric Mercer, and Neha Rungta. Vector-clock based partial order reduction for jpf. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, February 2014.

[134] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 53–68, Berlin, Heidelberg, 2013. Springer-Verlag.

[135] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 53–68, Berlin, Heidelberg, 2013. Springer-Verlag.

[136] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[137] Corneliu Popeea and Andrey Rybalchenko. Threader: A verifier for multi-threaded programs. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 633–636, Berlin, Heidelberg, 2013. Springer-Verlag.

[138] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *In TACAS*, pages 93–107. Springer, 2005.

[139] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[140] M. Ramalho, M. Freitas, F. Sousa, H. Marques, L. Cordeiro, and B. Fischer. Smt-based bounded model checking of c++ programs. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 147–156, April 2013.

[141] Thomson Reuters. Microsoft blames leap year for zune glitch. 2008. http://www.reuters.com/article/2009/01/01/us-microsoft-zune-idUSTRE5001LD20090101.

[142] P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NASA Formal Methods (NFM 2013)*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.

[143] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.

[144] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

[145] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: On hybrid approaches for ltl symbolic model checking. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 350–363, Berlin, Heidelberg, 2005. Springer-Verlag.

[146] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Jr. Hunt, WarrenA. and StevenD. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000.

[147] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[148] Matthew Staats and Mats Per Erik Heimdahl. Partial translation verification for untrusted code-generators. In Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki, editors, *ICFEM*, volume 5256 of *Lecture Notes in Computer Science*, pages 226–237. Springer, 2008.

[149] B. Steffen, M. Isberner, S. Naujokat, T. Margaria, and M. Geske. Property-Driven Benchmark Generation: Synthesizing Programs of Realistic Structure. STTT, To appear, London, UK, UK, 2014. Springer-Verlag.

[150] Bernhard Steffen. The rers grey box challenge 2012 — results. http://www.rers-challenge.org/2012/index.php?page=results#.

[151] Martin Sulzmann and Axel Zechner. Model checking dsl-generated c source code. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12, pages 241–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[152] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 42:230–265, 1936.

[153] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1995.

[154] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344. IEEE Computer Society, 1986.

[155] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[156] W. Visser. Personal communication.

[157] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

[158] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[159] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 382–396, Berlin, Heidelberg, 2008. Springer-Verlag.

[160] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[161] Zhenbo Xu, Jian Zhang, Zhongxing Xu, and Jiteng Wang. Canalyze: A static bug-finding tool for c programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 425–428, New York, NY, USA, 2014. ACM.

[162] Ivana Černá and Radek Pelánek. Distributed explicit fair cycle detection (set based approach). In Thomas Ball and SriramK. Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 49–73. Springer Berlin Heidelberg, 2003.