UNIVERSITY OF
Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems

by

Lucas Carvalho Cordeiro

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

in the

Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

April 2011

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Lucas Carvalho Cordeiro

Our reliance on the correct functioning of embedded systems is growing rapidly. Such systems are used in a wide range of applications such as airbag control systems, mobile phones, and high-end television sets. These systems are becoming more and more complex and require multi-core processors with scalable shared memory to meet the increasing computational power demands. The reliability of the embedded (distributed) software is thus a key issue in the system development. In this thesis we describe and evaluate an approach to reason accurately and effectively about large embedded software using bounded model checking (BMC) based on Satisfiability Modulo Theories (SMT) techniques. We present three major novel contributions. First, we extend the encodings from previous SMT-based bounded model checkers to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers and thus making our approach suitable to reason about embedded software. We then provide new encodings into existing SMT theories and we show that our translations from ANSI-C programs to SMT formulas are as precise as bit-accurate procedures based on Boolean Satisfiability. Second, we develop three related approaches for model checking multi-threaded software in embedded systems. In the lazy approach, we generate all possible interleavings and call the SMT solver on each of them individually, until we either find a bug, or have systematically explored all interleavings. In the schedule recording approach, we encode all possible interleavings into one single formula and then exploit the high speed of the SMT solvers. In the underapproximation and widening approach, we reduce the state space by abstracting the number of interleavings from the proofs of unsatisfiability generated by the SMT solvers. Finally, we describe and evaluate an approach to integrate our SMT-based BMC into the software engineering process by making the verification process incremental. In particular, our approach looks at the modifications suffered by the software system since its last verification, and submits them to a partly static and dynamic verification process, which is thus guided by a set of test cases for coverage. Experiments show that our SMT-based BMC can analyze larger problems and reduce the verification time compared to state-of-the-art techniques that use BMC, iterative context-bounding or counterexample-guided abstraction refinement.

Contents

Abbreviations xiii

Declaration Of Authorship xv

List of Publications xvii

Acknowledgements xix

1 Introduction 1
- 1.1 Problem Description . . . 4
- 1.2 Objectives . . . 7
- 1.3 Outline of the Solution . . . 8
- 1.4 Contributions . . . 11
- 1.5 Organization of the Thesis . . . 12

2 SAT-based and SMT-based Verification Techniques 15
- 2.1 Logical Foundations . . . 15
  - 2.1.1 Propositional Logic . . . 16
  - 2.1.2 Decision Procedures for Satisfiability . . . 18
  - 2.1.3 Satisfiability Modulo Theories . . . 21
  - 2.1.4 Linear-time Temporal logic . . . 25
- 2.2 Bounded Model Checking of Software . . . 29
  - 2.2.1 Formulation . . . 29
  - 2.2.2 Verification Conditions . . . 30
  - 2.2.3 Completeness . . . 32
    - 2.2.3.1 Craig Interpolation . . . 32
    - 2.2.3.2 K-Induction . . . 34
  - 2.2.4 BMC Architecture . . . 35
  - 2.2.5 Comparison to Other Verification Approaches . . . 36
- 2.3 Verification of Multi-threaded Systems . . . 39
  - 2.3.1 Concurrency and Interleaving . . . 40
  - 2.3.2 Partial Order Reduction Technique . . . 43
- 2.4 Summary . . . 46

3 SMT-based Bounded Model Checking for Embedded ANSI-C Software 49
- 3.1 Introduction . . . 49
- 3.2 SMT-based BMC Formulation . . . 51

3.3 Illustrative Example . . . 53
3.4 Encodings and Properties . . . 56
3.4.1 Scalar Data Types . . . 56
3.4.2 Fixed-Point Arithmetic . . . 58
3.4.3 Arithmetic Overflow and Underflow . . . 59
3.4.4 Arrays . . . 60
3.4.5 Structures and Unions . . . 61
3.4.6 Pointers . . . 63
3.4.7 Dynamic Memory Allocation . . . 66
3.5 Experimental Evaluation . . . 68
3.5.1 Experimental Setup . . . 68
3.5.2 Comparison of SMT solvers . . . 69
3.5.3 Error-Detection Capability . . . 72
3.5.4 Comparison to SMT-CBMC . . . 73
3.5.5 Comparison to CBMC . . . 74
3.6 Industrial Case Study . . . 75
3.7 Related Work . . . 78
3.8 Conclusions . . . 81

4 Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking 83
4.1 Introduction . . . 83
4.2 Preliminaries . . . 85
4.2.1 Multi-threaded Goto Programs . . . 85
4.2.2 Formal Model of Multi-threaded Software . . . 86
4.2.3 Context-Bounded Encoding . . . 88
4.3 Context-Bounded Model Checking of Multi-threaded Software . . . 89
4.3.1 Exploring the Reachability Tree . . . 89
4.3.2 Lazy Approach . . . 95
4.3.3 Schedule Recording Approach . . . 96
4.3.4 UW Approach . . . 98
4.3.5 Pruning the RT with Partial Order Reduction . . . 99
4.4 Verifying Race Conditions and Atomicity Violations . . . 103
4.4.1 Detecting Data Races . . . 103
4.4.2 Checking Atomicity . . . 104
4.5 Modelling Synchronization Primitives in Pthread . . . 105
4.5.1 Modelling Mutex Locking Operations . . . 106
4.5.2 Modelling Conditional Waiting . . . 108
4.6 Experimental Evaluation . . . 112
4.6.1 Comparison to MPOR and PPOR . . . 112
4.6.2 Comparison to CHESS . . . 114
4.6.3 Comparison to SATABS . . . 115
4.7 Related Work . . . 118
4.8 Conclusions . . . 119

5 Implementation of ESBMC 121
5.1 Introduction . . . 121

5.2 Tool Architecture . . . 122
5.3 Code Simplification and Reduction . . . 123
5.4 Exploiting Datatype Representations . . . 127
5.5 Evaluation of Performance Improvements . . . 128
5.6 Conclusions . . . 130

6 Integrating ESBMC into Software Engineering Practice 131
6.1 Introduction . . . 131
6.2 Continuous Verification . . . 132
6.3 Generalizing Test Cases . . . 135
6.4 Specifying Temporal Properties with Büchi Automata . . . 137
6.5 Experimental Evaluation . . . 140
6.5.1 Set-top Box Case Study . . . 141
6.5.2 Medical Device Case Study . . . 145
6.6 Related Work . . . 148
6.7 Conclusions . . . 149

7 Conclusions 151
7.1 Main Contributions . . . 153
7.2 Future Work Directions . . . 154
7.3 Concluding Remarks . . . 155

A ESBMC plug-in 157
A.1 Front-end Options . . . 157
A.2 BMC Options . . . 159
A.3 SMT Solver Configuration . . . 160
A.4 Property Check . . . 161
A.5 Concurrency Check . . . 162
A.6 Counterexample, Property Violation, and Claim Views . . . 163

B Static Analysis Benchmarks 165
B.1 EUREKA Suite . . . 165
B.2 POWERSTONE Suite . . . 165
B.3 NECLA Suite . . . 166
B.4 SNU-RT Suite . . . 167
B.5 VERISEC Suite . . . 169
B.6 WCET Suite . . . 176

C Functions of the Pthread Library 177

D Counterexample 179

References 185

List of Figures

1.1 A Synthetic Micro-benchmark. 6
1.2 Proposed SMT-based BMC procedure for software. 9

2.1 Syntax of the Background Theories 23
2.2 LTL semantics for the operators X, G, F, U, and R (when ψ first becomes *true* and when ψ never becomes *true*) over π [94]. 26
2.3 Example of a Kripke structure (with deadlock) for states s_0, s_1, and s_2 (where s_2 has a transition back to itself). 27
2.4 (a) A simple C program with a *for* loop. (b) The corresponding unwound C program of (a) converted into SSA form. 31
2.5 Computing image by interpolation [125]. 33
2.6 The CBMC Architecture. 35
2.7 (a) A C program with violated property. (b) The C program of (a) in SSA form. (c) Counterexample of C program in (a) 37
2.8 The CFG representation of threads T_A and T_B and we assume that initially the global variables a and b are set to zero, i.e., $a = 0$ and $b = 0$. . . . 41
2.9 The CFG that represents all possible interleaving sequences of threads T_A and T_B. 43
2.10 The transition system that represents the parallel execution of threads T_A and T_B. 44
2.11 Model context switches inside individual visible statements 44

3.1 ANSI-C program with two violated properties. 53
3.2 The program of Figure 3.1 in SSA form. 54
3.3 ANSI-C program with typecast from *char* to *int*. 58
3.4 Array out of bounds example. 61
3.5 ANSI-C program with union. 62
3.6 C program with pointer to an array. 64
3.7 C program with pointer to a struct. 65
3.8 A fragment of an ANSI-C program with dynamic memory allocation. . . . 67

4.1 Multi-threaded Goto Program Language 86
4.2 (a) A multi-threaded C program with an assertion violation. (b) The C program of (a) converted into multi-threaded goto form. 87
4.3 CFG of two threads of the goto program shown in Figure 4.2 (b). 88
4.4 Concurrent execution of two threads. 89
4.5 Fragment of the reachability tree of the multi-threaded goto-program of Figure 4.2(b). Nodes with dashed line represent program locations that violate the assertion statement in line 18 of Figure 4.2(b). 94

4.6 Algorithm of the lazy approach. 96
4.7 Schedule recording applied to the left-hand side of the RT in Figure 4.5. . 97
4.8 Algorithm of the UW approach. 99
4.9 (a) A simple multi-threaded C program. (b) The C program of (a) converted into goto form. 100
4.10 The reachability tree for threads t_1, t_2, and t_3 of the multi-threaded goto-program of Figure 4.9(b). Edges with dashed line represent transitions that can be eliminated by RW-POR. 101
4.11 The reachability tree for threads t_1, t_2, and t_3 after applying the RW-POR technique. 102
4.12 Modelling data race conditions for read operations (l = g). 104
4.13 Modelling data race conditions for write operations (g = l). 104
4.14 Modelling atomicity violation at visible statements. 105
4.15 Computation paths blocking on a mutex. 106
4.16 Modelling mutex lock operation. 107
4.17 An example of local deadlock with mutex on a database application. . . . 109
4.18 Modelling conditional waiting operation. 110
4.19 An example of deadlock with condition variable on a producer and consumer application. 111

5.1 Overview of the ESBMC architecture. 123
5.2 Code fragment of cyclic redundancy check. 124
5.3 *Goto*-program for the code fragment in Figure 5.2. 124
5.4 Loop unwound for the *goto*-program in Figure 5.3. 125
5.5 Code fragment of blit. 125
5.6 Code fragment of SumArray. 126
5.7 Code fragment of Fast Fourier Transformation. 126
5.8 A C program that uses shift-and-add to multiply two numbers. 127

6.1 Continuous Verification 133
6.2 (a) Original function to invert the sign of signal. (b) Optimized version. . 134
6.3 Implementation of a circular buffer. 136
6.4 A unit test for the functions shown in Figure 6.3. 136
6.5 The modified unit test for the test case shown in Figure 6.4. 137
6.6 Specifying Temporal Properties for Software. 139
6.7 The C-monitor thread to watch out for violations of the specified property.139
6.8 Event thread to model the hardware interrupt. 140
6.9 Concurrent execution of main, monitor and event threads. 141

A.1 Front-end options. 158
A.2 BMC options. 159
A.3 SMT Solver Configuration. 160
A.4 Property check. 161
A.5 Concurrency check. 162
A.6 Counterexample view. 163

List of Tables

2.1 Truth table. 17
2.2 Examples of First-Order Theories. 21

3.1 Definitions of ANSI-C types and their corresponding SMT representations. 57
3.2 Results of the comparison between CVC3, Boolector and Z3. Time-outs are represented with T in the Time column; Examples that exceed available memory are represented with M in the Time column. The subscript b indicates that the error occurred in the back-end. 70
3.3 Results of the error-detection capability of ESBMC. 72
3.4 Results of the comparison between ESBMC and SMT-CBMC [11]. 74
3.5 Results of the comparison between CBMC and ESBMC. Internal errors in the respective tool are represented with † in the Time column. The subscripts f and b indicate whether the errors occurred in the front-end or back-end, respectively. The superscript $*$ on the unwinding bound indicates that it is not large enough to prove or falsify the properties. . . 76
3.6 Results of the comparison between CBMC and ESBMC on a industrial case study. 79

4.1 Read-write analysis of interleaving equivalence between visible instructions. 103
4.2 Results of the comparison between MPOR and PPOR, and lazy, schedule, and UW ESBMC 113
4.3 Results of the comparison between ESBMC (v1.15.1) and Microsoft CHESS (v0.1.30626.0). 116
4.4 Results of the comparison between SATABS (v2.5) and ESBMC (v1.15.1). 117

6.1 Concrete values to check the circular buffer. 137
6.2 Transition function δ for the Büchi automaton shown in Figure 6.6. 138
6.3 Results for running the test cases for the functions *commandLoop* and *checkCommandParams*. 142
6.4 Results for checking the equivalence between the functions of the *exStbDemo* application. 143
6.5 Results of the LTL properties verification of the pulse oximeter. 146

B.1 Results of applying ESBMC to the verification of the benchmarks from the EUREKA suite. 166
B.2 Results of applying ESBMC to the verification of the benchmarks from the PowerStone suite. 166
B.3 Results of applying ESBMC to the verification of the correct benchmarks from the NECLA suite. 167

B.4 Results of applying ESBMC to the verification of the bad benchmarks from the NECLA suite. 168
B.5 Results of applying ESBMC to the verification of the benchmarks from the SNU-RT suite. 168
B.6 Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part I. 170
B.7 Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part II. 171
B.8 Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part III. 172
B.9 Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part I. 173
B.10 Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part II. 174
B.11 Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part III. 175
B.12 Results of applying ESBMC to the verification of the benchmarks from the WCET suite. 176

Abbreviations

ADC	Analog-to-Digital Converter
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
SAT	Boolean Satisfiability [24]
BMC	Bounded Model Checking [24]
BA	Buechi Automata
CBMC	C Bounded Model Checker [42]
CI	Continuous Integration [70]
CTL	Computational Tree Logic
CNF	Conjunctive Normal Form
CFG	Control Flow Graph [134]
CEGAR	Counterexample-Guided Abstraction Refinement [46]
DSP	Digital Signal Processor
ECTL	Existential Computational Tree Logic
DPLL	Davis-Putnam-Logemann-Loveland [24]
ECS	Effective Context Switches
EFSM	Extended Finite State Machine
ESBMC	Efficient SMT-Based Bounded Model Checker [52]
ESW	Embedded Software
FOL	First-Order Logic
FSM	Finite State Machine
HDL	Hardware Description Language
IC	Integrated Circuit
IF	Intermediate Frequency
IPC	Inter-Process Communication
LTL	Linear-time Temporal Logic
MDE	Model-Driven Engineering
MDG	Multiway Decision Graph
MPOR	Monotonic Partial Order Reduction [103]
OCL	Object Constraint Language
OFDM	Orthogonal Frequency-Division Multiplexing
PL	Propositional Logic [94]

POR	Partial Order Reduction [40]
PPOR	Peephole Partial Order Reduction [103]
PSL	Property Specification Language [7]
QF	Quantifier-Free Formula
QF_AUFBV	Quantifier-free formula over the theory of bit-vectors and bit-vector arrays with function and predicate symbols [164]
QF_AUFLIRA	Quantifier-free formula over closed linear formulas with function and predicate symbols over a theory of arrays of integer index and real value [164]
RT	Reachability Tree
RG	Region Graph
RTCTL	Real-Time Computational Tree Logic
SMT	Satisfiability Modulo Theories [19]
TA	Timed Automata
TECTL	Timed Existential Computational Tree Logic
TPN	Timed Petri Nets
TS	Transport Stream
UW	Under-approximation and Widening
VC	Verification Condition
VCG	Verification Condition Generator
WCET	Worst-Case Execution Time

Declaration Of Authorship

I, **Lucas Carvalho Cordeiro**, declare that this thesis entitled as **SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems** and the work presented in it are my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published.

Signed:

Date:

List of Publications

1. **Cordeiro, L.**, Fischer, B., Chen, H., and Marques-Silva, J. *Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints.* **In 6th Intl. Conf. on Embedded Software and Systems (ICESS)**, pp. 396-403, IEEE, 2009.

2. **Cordeiro, L.**, Fischer, B., and Marques-Silva, J. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software.* **In 24th Intl. Conf. on Automated Software Engineering (ASE)**, pp. 137-148, IEEE/ACM, 2009.

3. **Cordeiro, L.**, Fischer, B. and Marques-Silva, J. *Continuous Verification of Large Embedded Software using SMT-Based Bounded Model Checking.* **In 17th Intl. Conf. and Workshop on the Engineering of Computer Based Systems (ECBS)**, pp. 160-169, IEEE, 2010.

4. **Cordeiro, L.** *SMT-Based Bounded Model Checking for Multi-threaded Software in Embedded Systems.* **In 32nd Intl. Conf. on Software Engineering (ICSE), Doctoral Symposium**. pp. 373-376, ACM/IEEE, 2010.

5. **Cordeiro, L.** and Fischer, B. *Bounded Model Checking of Multi-threaded Software using SMT solvers.* **In 8th Intl. Workshop on Satisfiability Modulo Theories (SMT)**, Presentation-only paper, FLoC, 2010.

6. Rocha, H., **Cordeiro, L.**, Barreto, R., and Neto, J. *Exploiting Safety Properties in Bounded Model Checking for Test Cases Generation of C Programs.* **In 4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)**, pp. 121-130, SBC, 2010.

7. **Cordeiro, L.** and Fischer, B. *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking.* **To appear in 33rd Intl. Conf. on Software Engineering (ICSE)**, ACM/IEEE, 2011.

8. **Cordeiro, L.**, Fischer, B., and Marques-Silva, J. *SMT-Based Bounded Model Checking for Embedded ANSI-C Software.* **Under review in the IEEE Transactions on Software Engineering (TSE)**, IEEE, 2011.

Acknowledgements

I would like to thank my friend and supervisor Dr. Bernd Fischer. I could not have completed this thesis without his invaluable support, guidance, encouragement and friendship over the years of my research. Beyond any duty, his effort helped me considerably to turn this work into a PhD dissertation. I would also like to thank Prof. Joao Marques-Silva and Prof. Michael J. Butler for their objective advice and assistance, and Prof. Mark Zwolinski and Prof. Rupak Majumdar for agreeing to be the examiners of my work. Many thanks to my friends and colleagues who helped me not only with fruitful discussions about my work and "proof-reading" my scribbles, but also with my personal life (beyond the work). I would also like to thank my sponsors (ORSAS and ECS) for their financial support. Last but not least, I would like to thank my wife for her unconditional love and support. Without her continuous encouragement, I certainly would not be where I am today. I am also very grateful to my parents for their support in my education and my desire to always learn more and more.

To my dearest wife and son . . .

Chapter 1

Introduction

Embedded computer systems are used in a variety of sophisticated applications, which range from safety-critical systems such as nuclear reactors and automotive controllers, to entertainment software such as games and graphics animation. Embedded systems are ubiquitous in modern day information systems and are becoming increasingly important in our society. As a consequence, human life has also become more and more dependent on the services provided by this type of system. In general, an embedded system may be viewed as consisting of an electrical and mechanical subsystem, a controlling embedded computer and a man-machine interface, which together perform a group of dedicated functions within a larger system [104]. More specifically, it consists of a set of hardware/software components that together implement a set of functionalities while satisfying constraints such as timing, power dissipation, and monetary costs.

Embedded systems also replace many mechanical and hydraulic control systems within safety-critical and high dependability applications. Despite the criticality of the applications, the main role of the software in embedded systems is the interaction with the physical world, rather than the transformation of data. Embedded software has thus a number of characteristics that differ substantially from conventional desktop applications. For example, it is dedicated to perform a particular task, which requires to meet the timing constraints of the application, access the memory region, handle concurrency, and control the hardware registers. The reliability of the software in embedded systems then plays an important role to avoid catastrophic errors (especially in safety-critical systems) and to reduce costs (because software errors are expensive [1]).

Due to the high pressure imposed by the market to launch new products, coupled with evolving system specifications, semiconductor and system development companies are forced to choose flexible implementations where new products can be built quickly [51]. The increasing computational power and decreasing size and cost of processors is enabling system designers to move more functionalities to software. Market analysis shows that software-based implementations account for more than 80% of system development

in the embedded systems domain [174]. The increasing number of functionalities moved to software-based implementations leads to difficulties in verifying design correctness. In practice, however, this verification is of importance due to the dependability properties (briefly reliability and availability) required in several embedded system domains such as automotive, industrial automation, and transportation. In order to verify the design correctness of hardware blocks, model checking has been widely used as a verification methodology [47]. However, the verification of embedded software has always been difficult, mainly due to the stringent constraints imposed by the hardware (e.g., real-time, memory allocation, interrupts, and concurrency) when verifying the design correctness.

Nowadays, *peer reviewing* and *testing* are the major software verification techniques used in practice [14]. A peer review is carried out by a team of experienced software engineers that inspects the software and preferably has not been involved in the development of the software under review. Empirical studies show that the peer review technique is able to catch between 31% and 93% of the defects with a median around 60% [14]. Software testing, as opposed to peer review, is a dynamic technique that actually runs the software instead of analyzing it statically without executing. Correctness is thus determined by forcing the software to traverse a set of execution paths and by observing during test execution the actual and expected output of the software. These approaches, however, take up to 70% of the total development time to find out bugs and implement the necessary corrections in the design [79].

We can thus see that there is clearly a tradeoff between time-to-market (i.e., the time between product conception and arrival on the market), costs and quality. On the one hand, consumer electronics companies strive to shorten the time-to-market with the purpose of being the first one to launch the product and maximize the profit. However, some steps of the development process might be skipped to achieve this, thus compromising quality. On the other hand, software bugs cause a loss of billions of US dollars annually [1]. It is then of great importance to detect software bugs as early as possible with minimum effort, cost and time, because the cost of repairing a software flaw during maintenance is hundreds of times more expensive than a fix in an early design phase. Consequently, the development of techniques to ensure low-defect embedded systems given their complexity (i.e., size and shorter development time) represents a significant research challenge, a challenge that has increased significantly with the emergence of multi-threaded applications. The motivation of this thesis is thus to deal with the increasing difficulty in verifying embedded systems design correctness within the market window and with the required level of confidence in the signed-off design.

Bounded model checking (BMC) based on Boolean Satisfiability (SAT) has already been successfully applied to verify sequential software in embedded systems and discover subtle errors in real designs [24]. However, the verification of multi-threaded software is a hard problem, because we need to explicitly account for interleavings[1] of transitions

[1] An interleaving represents a possible execution of the program where all of the concurrent events

of different threads. A major strength of BMC to combat this problem is that BMC analyzes only bounded program runs (thereby achieving decidability) and state space reduction is exploited internally by state-of-the-art SAT or SMT (Satisfiability Modulo Theories) solvers with the use of conflict clauses and non-chronological backtracking. The basic idea of the BMC technique is thus to check (the negation of) a given property at a given depth: given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a verification condition (VC) ψ such that ψ is satisfiable if and only if ϕ has a counterexample of depth less than or equal to k. Standard Boolean Satisfiability solvers can be used to check whether ψ is satisfiable. In BMC of software, the bound k limits the number of loop iterations and recursive calls in the program. BMC of software thus generates VCs that reflect the exact path in which a statement is executed, the context in which a given function is called, and the bit-accurate representation of the expressions. Proving the validity of the VCs arising from (sequential or) multi-threaded software remains a major performance bottleneck in verifying embedded software, despite attempts to cope with increasing system complexity by applying SMT solvers.

In this thesis, we develop and evaluate approaches that exploit the use of SMT solvers for model checking multi-threaded ANSI-C software and our modelling of the synchronization primitives of the Pthread library [135]. We also describe translations from ANSI-C programs to SMT formulas with the same precision as bit-accurate SAT-based procedures. In this sense, we extend the encodings from previous SMT-based bounded model checkers [11, 71] to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers. In contrast to previous fully symbolic approaches to handle multi-threaded systems (e.g., [73, 102, 103, 152, 84]), we combine symbolic model checking with explicit state space exploration. We analyze a reachability tree, which is a description of all reachable states of a program, built by unfolding the actions of each thread and we then propose novel exploration methods to traverse this reachability tree. In particular, we explicitly explore the possible interleavings of a program (up to the given context bound) while we treat each interleaving itself symbolically. We also exploit SMT techniques to prune the property and data dependent search space and to remove interleavings that are not relevant by analyzing the proof of unsatisfiability.

In summary, we propose a comprehensive SMT-based context-bounded model checking procedure which we implemented in the ESBMC (Efficient SMT-based Context-Bounded Model Checker)[2] tool for verifying multi-threaded software in embedded systems written in ANSI-C. In our work, we consider embedded software because it has characteristics that make it attractive for BMC, e.g., dynamic memory allocations and recursion are highly discouraged, and that make the limitations of *bounded* model checking less stringent. We also chose ANSI-C because it is the most common implementation language

are arranged in a linear order.

[2] Available at http://users.ecs.soton.ac.uk/lcc08r/esbmc/

for embedded software (and in particular for developing optimized applications), but all techniques that we describe in this thesis are also applicable to languages that are similar to ANSI-C (e.g., MISRA-C). Our experimental results show that our approach scales significantly better than both SAT-based and SMT-based versions of the CBMC model checker [42, 105] and SMT-CBMC [11], a bounded model checker for sequential C programs that is based on the SMT solvers CVC3 [20] and Yices [65]. We also show that our approaches to verify multi-threaded software can analyze larger problems and substantially reduce the verification time compared to state-of-the-art techniques for multi-threaded verification that use BMC (e.g., [103]), iterative context-bounding algorithms (e.g., [138]) and others that implement counterexample-guided abstraction refinement (CEGAR) techniques (e.g., [44]).

The rest of this chapter describes the problem statement, objectives and outlines the solution. It then summarises our contributions and presents the structure of the thesis.

1.1 Problem Description

This PhD thesis tackles two major problems in computer-aided verification: (1) providing suitable encodings into the SMT theories to reason accurately and effectively about realistic embedded programs and (2) exploiting SMT techniques to leverage bounded model checking of multi-threaded software.

Part of the first problem stems from the fact that most software verification tools are unable to reason accurately about embedded programs. Most programming languages provide basic data types that have a bounded range defined by the number of bits allocated to each of them. They also contain constructs such as structures, unions, and pointers that are not directly supported by the SMT solvers, and are often encoded imprecisely using axioms and uninterpreted functions by software verification tools that employ theorem provers (e.g., Simplify [62]) as back-end (e.g., ESC/Java [69], BLAST [89], and Magic [35]). Nevertheless, in order to reason about embedded software accurately, an SMT-based software verification tool must consider a number of issues that are not easily mapped into the theories supported by the SMT solvers, e.g., QF_AUFBV (the theory of bit-vectors and bitvector arrays with function and predicate symbols) and QF_AUFLIRA (closed linear formulas with function and predicate symbols over a theory of arrays of integer index and real value) [164]. In previous work on SMT-based BMC for software [11, 71] only the theories of uninterpreted functions, arrays and linear arithmetic were considered, but no encoding was provided for ANSI-C [95] constructs such as bit operations, unions, fixed-point arithmetic, pointers (e.g., pointer arithmetic and comparisons) and dynamic memory allocation. This limits its usefulness for analyzing and verifying realistic embedded software written in ANSI-C.

The other part of the first problem stems from the fact that most software verification

tools are unable to reason effectively about embedded programs. There are tools that employ SAT solvers as back-end and thus provide a bit-level accurate symbolic simulator (e.g., CBMC [42], F-SOFT [96]), but they have limitations due to inefficient translations and loss of high-level design information during the BMC problem formulation, especially when reasoning on the propositional encoding of arithmetical operators (e.g., multiplication) [49]. SMT solvers, however, often integrate a simplifier, which applies standard algebraic reduction rules (e.g., $a \wedge \mathit{false} \mapsto \mathit{false}$) and contextual simplification (e.g., $b = 7 \wedge p(b) \mapsto b = 7 \wedge p(7)$) before *bit-blasting* or *bit-flattening* (i.e., replacing the word-level operators by bit-level circuit equivalents) propositional expressions to a SAT solver. As structural word-level information remains in the problem formulation, bit-blasting is used by the SMT solvers only as a last resort if the more abstract and less expensive techniques are not powerful enough to solve the problem at hand (e.g., the incremental and layered approach which permits strengthening incrementally the model of the arithmetic operators [28]).

Consequently, new encodings are needed into existing SMT theories in order to make verification scalable and to model precisely ANSI-C scalar data types (with accurate arithmetic over- and underflow), arrays, pointers, structures, and unions.

Second, the widespread use of multi-core processors with scalable shared memory in embedded systems is already having a tangible impact on development and testing for major software vendors [144]. However, the verification of the software design and the correctness of its multi-threaded implementations has become increasingly difficult, for at least three reasons. The first reason is that the verification of multi-threaded programs exhibits more non-deterministic behaviour (i.e., the choice of interleaving among threads in addition to the non-deterministically chosen values), which results in a large state space that must be explored by a model checker. The second reason is that concurrency errors are tricky to reproduce and debug because they usually occur under specific thread interleavings. These errors most frequently manifest as deadlock, data races, atomicity violations, and order violations and finding them in realistic multi-threaded programs is challenging. In particular, an empirical study shows that the most common concurrency errors are related to atomicity and order violations (approx. 67%) and deadlock (approx. 30%) [117]. This leads to the third reason namely that errors related to multi-threaded software typically involve changes in program state due to particular interleavings of multiple threads of execution, thus making them difficult to understand in the code.

As an example of the non-determinism related to the choice of interleaving among threads, we consider a synthetic micro-benchmark extracted from Ghafari et al. [74], which checks for a single valid property as shown in Figure 1.1. This micro-benchmark is used in [74] to check the scalability of multi-threaded software verification tools by varying two key problem parameters: the number of threads (n) and program statements (s).

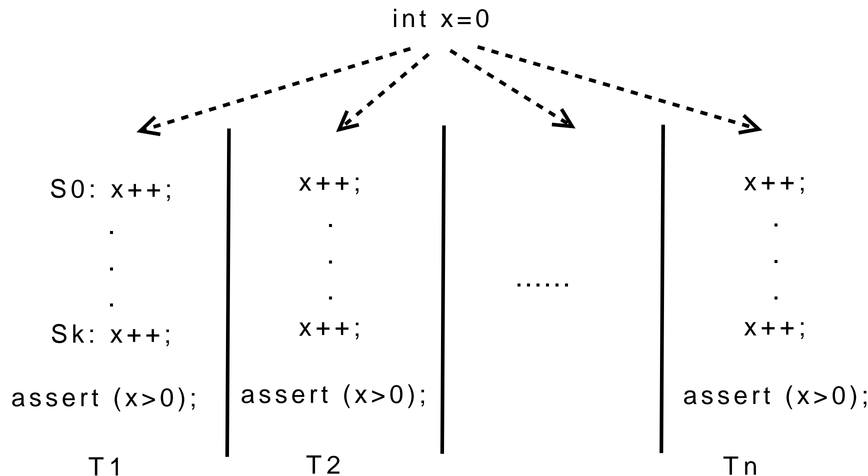

FIGURE 1.1: A Synthetic Micro-benchmark.

This micro-benchmark uses a shared global variable x, which in the initial state is initialized to 0. Then, n threads are created such that each thread consists of s increments of the variable x followed by an assertion that checks if x is greater than 0. Although this micro-benchmark is a simplistic example of a multi-threaded program, it has essentially three key elements that make it worthwhile to mention: local state (the program counters), shared global state x, and long data-dependency chains that grow with code size and must then be inspected to prove the assertions. Therefore, this micro-benchmark shows that as we increase n and s (and consequently the data-dependency), the number of interleavings can grow very quickly (i.e., the number of possible execution sequences is $O(n^s)$) since context switches among threads (due to the global variable x) increase the number of possible execution paths considerably. Hence, in order to fully verify multi-threaded programs against a given specification, all possible interleavings must be considered, and this thus represents a challenging problem in computer-aided verification.

Recently, there have been attempts to extend BMC to the verification of multi-threaded software [73, 102, 103, 152]. The main challenge remains the classic state space explosion problem in which the number of interleavings grows exponentially with the number of threads and program statements as sketched above. Previous attempts are unable to model check realistic multi-threaded programs (e.g., [152] evaluate their approach on a concurrent bubblesort and [103] on a parameterized version of the dining philosophers model, which are untypical multi-threaded C programs.) and they are unable to find bugs related to local and global deadlock (e.g., [73, 102, 103, 152]). Other attempts (e.g., [39]) encode the semantics of the SystemC scheduler, which does not allow preempting a thread at any visible instruction in its execution and it is thus unsuitable to model check multi-threaded software.

As far as we are aware, there is no other work that considers a comprehensive SMT-based

context-bounded model checking technique to verify real-world multi-threaded ANSI-C software by combining symbolic model checking with explicit state space exploration. Thus the problem considered in this thesis is expressed in the following question: *can an algorithmic method reason accurately about realistic multi-threaded software in embedded systems and at the same time control the verification complexity?*

1.2 Objectives

The main objective of this thesis is thus to propose and evaluate *an SMT-based bounded model checking formulation to reason accurately about multi-threaded software, for example, used in embedded systems.* In particular, we focus on embedded applications written in ANSI-C that are platform-independent (single- and multi-threaded); and we do not model check platform-dependent software (e.g., software that controls the hardware registers) nor the timing constraints of the application. We further try to exploit the SMT solvers to remove possible undesired models of the system in order to satisfy a given property. In this respect, we develop new algorithmic methods and corresponding tools based on SMT techniques to verify single- and multi-threaded software (with shared variables) in embedded systems. More specifically we will:

1. Provide details of an accurate translation from programs written in (full) ANSI-C into quantifier-free (QF) first-order logic formulae (cf. Chapter 3).

2. Propose approaches to model check multi-threaded software with shared variables by combining symbolic model checking with explicit state space exploration and by bounding the number of context switches allowed among threads (cf. Chapter 4).

3. Develop heuristics to simplify the unwound formula arising from BMC instances and exploit the different theories and SMT solvers (cf. Chapter 5).

4. Detect design errors and integration problems as quickly as possible by exploiting information from the software configuration management (SCM) system (cf. Chapter 6).

In Chapter 3, we propose a new encoding for (full) ANSI-C by exploiting the background theories supported by the SMT solvers (e.g., uninterpreted functions, arithmetic, bit-vectors, and arrays). Hence, we extend and combine these background theories to develop an approach to model precisely the ANSI-C program's semantics. We will demonstrate that this new encoding allows us to reason accurately about realistic embedded software systems and improve the performance of software model checking for a wide range of applications.

In Chapter 4, we describe and evaluate three approaches to SMT-based bounded model checking: lazy, schedule recording and underapproximation and widening. In all three approaches, we combine symbolic model checking with explicit state space exploration by constructing a reachability tree derived from the program and we also use a context-bounded analysis [112, 171] that limits the number of context switches it explores. This thus allows exploring explicitly the possible thread interleavings (up to the given context bound) while treating each interleaving itself symbolically. We will evaluate our approaches over several multi-threaded applications and show that they substantially reduce the verification time compared to other state-of-the-art techniques.

In Chapter 5, we exploit the different background theories of SMT solvers and combine different theories and solvers, based on an analysis of the syntactic structure of a given ANSI-C program. This allows exploiting the structure provided by the program, and thus, improving scalability by making the analysis computationally more tractable. Additionally, we describe a set of simplifications that we used in order to reduce the unwound formula. We will evaluate the performance improvement of these simplifications and heuristics over a large set of benchmarks and show that they prevent overburdening the model checker in realistic applications.

In Chapter 6, we describe an approach to integrate our SMT-based model checker into the software engineering practice by focusing systematically the verification effort on new or modified functions. We investigate the use of equivalence checking to determine whether modified functions need to be re-verified formally and use existing test cases to reduce the search space for the model checker, thus combining dynamic and static verification. We will demonstrate through case studies that the proposed approach can potentially improve the error-detection capability and reduce the overall verification time.

1.3 Outline of the Solution

Our approach deals with the theoretical and pragmatic aspects of using SMT techniques to model check single- and multi-threaded software in embedded systems. We thus develop algorithms and the corresponding tools and evaluate them using standard software model checking benchmarks. The tools are built using a number of advanced (and complex) techniques, including symbolic execution engines, satisfiability modulo theories solvers, context-bounded analysis, and partial order reduction. We use off-the-shelf software wherever possible and focus our effort on a comprehensive and implemented SMT-based model checking procedure to verify embedded software or more precisely, finite approximations of embedded software. In particular, we reuse the C/C++ front-end from the CProver framework[3] and use existing SMT solvers.

[3] The CProver framework consists of the components on which the verification tools CBMC [42] and SATABS [43] are based. It provides a mature and robust front-end for ANSI-C and C++ programs.

Figure 1.2 shows an overview of our SMT-based bounded model checking procedure for single- and multi-threaded software in embedded systems. In Figure 1.2, the box labelled *CFG* with solid lines represents the component that we reused without any modification from the CProver framework (i.e., the construction of the control-flow graph). The boxes labelled *BMC* (i.e., symbolic execution engine) and *properties* (i.e., property instrumentation) that are shown with thick dashed lines represent the components that we substantially extended from the CProver framework in order to simplify the unwound formula and handle multi-threaded programs; in particular, the CProver framework does not perform bounded model checking of multi-threaded programs. The boxes labelled *scheduler* and *verification conditions* with thick solid lines represent components that we developed from scratch. We describe here only a summary of each phase of our proposed approach; more details are presented in the next chapters. The phases can be described as follows:

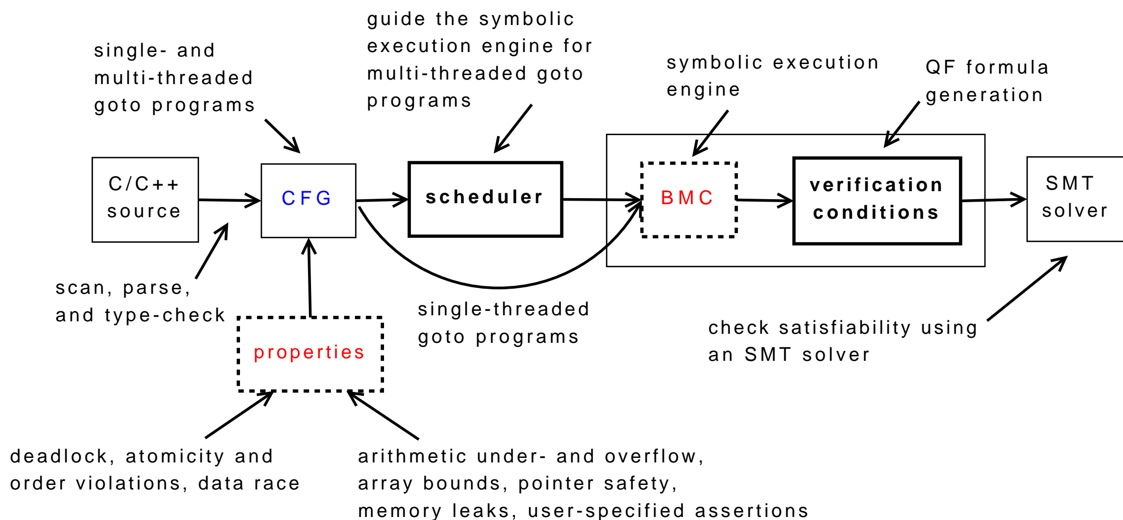

FIGURE 1.2: Proposed SMT-based BMC procedure for software.

- **Building the control-flow graph (CFG):** In BMC, the program to be analyzed is modelled as a state transition system, which is extracted from the control-flow graph (CFG) [134]. The CFG is used as part of a translation process from program text to single static assignment (SSA) form. This component is thus responsible for scanning, parsing, and type-checking the C/C++ code and is reused from the CProver framework without any modification.

- **Automatic generation of (concurrency) properties:** In addition to the language-specific safety properties that are generated automatically by the CProver framework (e.g., absence of arithmetic under- and overflow, out-of-bounds array indexing, or NULL-pointer dereferencing), we extend its class of properties to generate verification conditions to check for memory leaks, data races and atomicity and order violations in single- and multi-threaded programs. We also provide a new

instrumented model of the Pthread functions to generate verification conditions to check for local and global deadlocks in the client code.

- **Thread scheduler:** This component guides the symbolic execution between threads and systematically explores all the possible interleavings. To this end, we construct a reachability tree (RT) of a multi-threaded program by unwinding the control-flow graph in a depth-first search manner. We thus generate explicitly the thread interleavings (with techniques similar to explicit-state model checking) and we then guide the symbolic execution engine in order to encode each thread symbolically. In particular, we explore the reachability tree by using three different approaches called *lazy exploration*, *schedule recording*, and *underapproximation and widening*. In the *lazy exploration* approach, we traverse the RT depth-first, and simply call the single-threaded BMC procedure on the interleaving whenever we reach an RT leaf node. We stop the RT traversal either when we find a bug, or have systematically explored all interleavings. In the *scheduling recording* approach, we use the RT to encode all the possible execution paths into one single formula, which is then fed into the SMT solver. In the *underapproximation and widening* approach, we model check models with an increasing set of allowed interleavings. We start from an underapproximation describing a single interleaving and widen the model by adding more interleavings incrementally based on the proof objects generated from an SMT solver.

- **Symbolic execution engine:** For single-threaded programs, this component takes as input the CFG representation of the program, a property ϕ, and a bound k. It derives as output a verification condition ψ_k such that ψ_k is satisfiable if and only if ϕ has a counter-example of length k or less. For multi-threaded programs, this component takes as input a reachability tree $\Upsilon = \{\nu_1, \ldots, \nu_N\}$ (where ν_i is a given node in the reachability tree) that represents the program unfolding for a context bound C and a bound k, and a property ϕ. It derives as output a verification condition ψ_k^{π} for a set of interleavings $\pi = \bigwedge_{i=0}^{m} \pi_m$ (where m is the total number of interleavings) or for a given interleaving (or computation path) $\pi_i = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^{π} (or $\psi_k^{\pi_i}$) is satisfiable if and only if ϕ has a counterexample of depth less than or equal to k that is exhibited by π (or π_i). Here, we extend the SSA form of the symbolic execution engine to avoid naming conflicts (when verifying multi-threaded programs) such as *local* (i.e., threads that contain local variables with the same name) and *path* (i.e., nodes of the RT that contain variables with the same name) conflicts. Additionally, we implement a set of simplification techniques (e.g., constant propagation and forward substitution) to reduce the unwound formula and we also perform an up-front analysis in the control-flow graph of the program during the symbolic execution to determine the most appropriate encoding and solver for a particular program.

- **Quantifier-free formula generation:** This component takes as input the veri-

fication conditions generated by the symbolic execution engine and encodes them into a quantifier-free formula in a decidable subset of first-order logic. Here, new encodings are provided into existing SMT theories to model precisely ANSI-C scalar data types (with accurate arithmetic overflow and underflow), arrays and pointers (i.e., pointer arithmetic and comparisons), structures and unions, memory allocation and fixed-point arithmetic.

1.4 Contributions

The main contribution of this PhD thesis is the development, implementation, and evaluation of a comprehensive SMT-based bounded model checking procedure to verify realistic single- and multi-threaded software in embedded systems. In this respect, this thesis makes three major novel contributions.

First, we describe the details of an accurate translation from ANSI-C programs into quantifier-free formulae using the SMT logics QF_AUFBV and QF_AUFLIRA from the SMT-LIB and we also apply a set of optimization techniques to prevent overburdening the solver. We demonstrate that our encoding and optimizations improve the performance of software model checking for a wide range of embedded software systems. Additionally, we show that our encoding allows us to reason about arithmetic under- and overflow, pointer safety, memory leaks, array bounds, atomicity and order violations, deadlock, data race, and user-specified assertions; and to verify programs that make use of bit-level, pointers, dynamic memory allocation, structs, unions and fixed-point arithmetic. Note that we do not require the user to annotate the programs with pre/post-conditions and the verification is thus completely automatic. We also use three different SMT solvers (CVC3 [20], Boolector [31], and Z3 [57]) in order to check the effectiveness of our encoding techniques. We considered these solvers because they were the most efficient ones for the categories of QF_AUFBV and QF_AUFLIRA in the last SMT competitions.[4] As far as we are aware, no SMT-based bounded model checking tool existed that can reliably handle full ANSI-C. We also exploit different background theories and solvers, based on an analysis of the syntactic structure of a given ANSI-C program in order to improve scalability and precision in a completely automatic way. To the best of our knowledge, this is the first work that reasons accurately about ANSI-C constructs commonly found in embedded software and extensively applies SMT solvers to check the verification conditions emerging from the bounded model checking of embedded software industrial applications.

Second, we exploit SMT to improve bounded model checking of multi-threaded software. In particular, we exploit SMT solvers to prune the property and data dependent search space (via non-chronological backtracking and conflict clauses learning) and to remove

[4]The results are available at http://www.smtcomp.org

possible undesired models (i.e., interleavings that are not relevant) of the system in order to satisfy a given property (which is done by analyzing the proof of unsatisfiability). We describe and evaluate three approaches: lazy, schedule recording, and underapproximation and widening (UW) to model check multi-threaded software with shared variables and locks using bounded model checking based on SMT techniques and our modelling of the synchronization primitives of the Pthread library. Here, the main novelty is in the combination of symbolic model checking with explicit state space exploration that underlies all three approaches. To the best of our knowledge, the lazy approach has not been described or evaluated in the literature. Similarly, underapproximation and widening has not been used for bounded model checking of multi-threaded software. Additionally, our approach is based on the new notion of effective context switches (ECS) blocks and it thus uses a different encoding from Grumberg et al. [84]. The difference between our schedule recording and the approaches proposed by [73, 102, 103, 152] is that they all work in a fully symbolic context. We also describe a new modelling of the Pthread synchronization primitives for mutex and condition variables that allows us to detect local and global deadlock.

Finally, we explore a new concept called continuous verification to detect design errors and integration problems as quickly as possible by exploiting information from the software configuration management (SCM) system, systematically focusing the verification effort on new or modified functions [54]. We thus add a state-space reduction technique for our SMT-based bounded model checking procedure, which looks at the modifications suffered by the system since its last verification, and submits them to a partly static, partly dynamic "continuous" verification process, guided by a set of test cases for coverage. As a result, we integrate the continuous verification approach with the combination of different encodings and solvers in order to allow us to verify larger parts of the state space of the system (compared to software model checkers only) and explore more exhaustively the state space (compared to testing only).

1.5 Organization of the Thesis

This introduction has outlined the context, motivation, and problem addressed by this thesis, and the objectives, solution and contributions of the research. The remainder of the chapters of this thesis are organized as follows:

Chapter 2, *SAT-based and SMT-based Verification Techniques*, overviews the main concepts needed to understand this thesis, such as propositional logic, SAT-based bounded model checking, satisfiability modulo theories and concurrent systems and reviews some methods to achieve completeness in the BMC framework such as Craig interpolation and k-induction. It also includes an explanation about verification conditions and partial-order reduction. Additionally, this chapter reviews the related work on model checking

sequential and multi-threaded software as well as techniques applied to the verification of large embedded software systems.

Chapter 3, *SMT-based Bounded Model Checking for Embedded ANSI-C Software*, describes the encoding and application of different background theories and SMT solvers to the verification of embedded software written in ANSI-C in order to improve scalability and precision in a completely automatic way. We evaluate these approaches on both standard software model checking benchmarks and typical embedded software applications from telecommunications, control systems, and medical devices. Our experiments show that our approaches can analyze larger problems than existing tools and substantially reduce the verification time.

Chapter 4, *Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking*, describes and evaluates three approaches to model check multi-threaded software with shared variables and locks using bounded model checking based on Satisfiability Modulo Theories (SMT) and our modelling of the synchronization primitives of the Pthread library. In all three approaches, we bound the number of context switches allowed among threads in order to reduce the number of interleavings explored. This chapter shows that our approaches can analyze larger problems and substantially reduce the verification time compared to state-of-the-art techniques that use BMC, iterative context-bounding algorithms or counter-example guided abstraction refinement.

Chapter 5, *Implementation of ESBMC*, describes the main software components of the ESBMC architecture and the simplifications that we used in order to reduce the unwound formula. It also evaluates the simplification techniques, which give a substantial performance improvement over a large set of benchmarks.

Chapter 6, *Integrating ESBMC into Software Engineering Practice*, describes a new approach called *continuous verification* to detect design errors as quickly as possible by exploiting information from the software configuration management system and by combining dynamic and static verification to reduce the state space to be explored. This chapter shows that the proposed approach can potentially reduce the overall verification time in a case study from the telecommunications domain.

Finally, Chapter 7, *Conclusions*, concludes the contributions of this thesis and describes how our work differ from the others. This chapter also outlines the limitations of our approaches and presents some directions for future work.

Appendix A describes an Eclipse plug-in for the ESBMC model checker that can assist the software engineer during the verification process. This plug-in was developed with the help of Qiang Li during his summer internship.

Appendix B shows the detailed results of the error-detection capability of ESBMC over a large set of well-known static analysis benchmarks.

Appendix C describes the main functions of the POSIX Pthread library [135] that ES-BMC supports.

Appendix D shows an example of the counterexample that is generated by ESBMC for a multi-threaded program.

Chapter 2

SAT-based and SMT-based Verification Techniques

This chapter introduces the main concepts needed to understand this thesis. It is divided into three main sections. The first section, *Logical Foundations*, defines the syntax and semantics of propositional logic and sketches decision procedures for checking satisfiability of propositional formulae. It also describes the background theories of the Satisfiability Modulo Theories (SMT) solvers that are used throughout this thesis, and how to specify safety and liveness properties using linear-time temporal logic. The second section, *Bounded Model Checking of Software*, presents the BMC technique and shows how to achieve completeness in BMC via Craig interpolation and k-induction techniques. This section also overviews the BMC architectures used in software verification and compares the BMC technique to other software verification approaches that also use logic to describe states and transformations between system states. Finally, the third section, *Verification of Multi-threaded Systems*, presents concepts and definitions of multi-threaded (concurrent) systems and the partial order reduction technique used to prune the state space of multi-threaded systems.

2.1 Logical Foundations

Logic can be defined by means of symbols and a system of rules to manipulate the symbols [29]. The use of logic allows us to model the programs and to reason about them formally. This section thus introduces the logical foundations that will be the basis for the explanation of our techniques described in Chapters 3, 4, and 6.

2.1.1 Propositional Logic

This section recalls the definition of propositional logic (PL) syntax and semantics along with some examples. Further information can be found in textbooks [29, 94, 109, 130]. The syntax of PL consists of symbols and rules so that we can combine the symbols to construct "sentences" (more specifically formulae). Generally speaking, propositional logic or calculus is a two-valued logic, which is based on the assumption that every sentence is either true or false. A truth value (or a logical value, which is represented by *tt* or *ff*), is a value indicating the relation of a proposition (i.e., the meaning of the sentence) to truth. The basic elements of PL are the constants *true* (sometimes also represented as $\top$ or 1) and *false* (sometimes also represented as $\bot$ or 0) and the propositional variables: $x_1, x_2, \ldots, x_n$ (whose set is usually denoted by the letter X, except where noted otherwise and n is a finite number of propositional variables). Logical operators (e.g., $\neg, \wedge$), also called Boolean operators, provide the expressive power of PL.

Definition 2.1. *The syntax of formulae in PL is defined by the following grammar:*

$$\begin{aligned} Fml &::= Fml \wedge Fml \mid \neg Fml \mid (Fml) \mid Atom \\ Atom &::= Variable \mid true \mid false \end{aligned}$$

Using the logical operators conjunction ($\wedge$) and negation ($\neg$), the full power of propositional logic is obtained. Other logical operators such as disjunction ($\vee$), implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), exclusive or ($\oplus$), and conditional expression (*ite*) can be defined as follows.

Definition 2.2. *We define the usual logical operators as follows:*

- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$
- $\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$
- $\phi_1 \oplus \phi_2 \equiv (\phi_1 \wedge \neg\phi_2) \vee (\phi_2 \wedge \neg\phi_1)$
- $ite\,(\theta, \phi_1, \phi_2) \equiv (\theta \wedge \phi_1) \vee (\neg\theta \wedge \phi_2)$

A PL formula is then defined in terms of the basic elements *true*, *false*, or a propositional variable x; or the application of one of the following logical operators to a formula ϕ: "not" ($\neg\phi$), "and" ($\phi_1 \wedge \phi_2$), "or" ($\phi_1 \vee \phi_2$), "implies" ($\phi_1 \Rightarrow \phi_2$), "iff" ($\phi_1 \Leftrightarrow \phi_2$). "parity" ($\phi_1 \oplus \phi_2$) or "ite" ($ite\,(\theta, \phi_1, \phi_2)$).

Each operator in PL has an arity (i.e., the number of arguments that it takes). The operator "not" is unary while the other operators are binary, except for "ite", which is

a ternary operator. The left and right arguments of $\Rightarrow$ are called the antecedent and consequent respectively. The propositional variables, and propositional constants, *true* and *false*, stand for indecomposable propositions, known as *atoms*, or *atomic propositions*. A literal is an atom β or its negation $\neg\beta$. A formula is a literal or the application of a logical operator to a formula or formulae.

Formulae in PL are strings over the alphabet $\{x_1, x_2, x_3, \ldots\} \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\} \cup \{(,)\}$. The string $\wedge(\neg) \vee x_1 x_2 \Leftrightarrow$ is a word over that alphabet, but it does have any meaning as far as propositional logic is concerned.

Definition 2.3. *We say that a PL formula is a well-formed formula if we use the construction rules from Definition 2.1 to obtain it given that negation has priority over conjunction.*

Definition 2.4. *We define the relative precedence of the logical operators from highest to lowest as follows:* $\neg$, $\wedge$, $\vee$, $\Rightarrow$ *and* $\Leftrightarrow$.

In order to check whether a given PL formula is *true* or *false*, we first define a mechanism for evaluating the propositional variables by means of *interpretations*. An interpretation I assigns to every propositional variable exactly one truth value. For instance, $I = \{x_1 \mapsto tt, x_2 \mapsto ff\}$ is an interpretation assigning *true* to x_1 and *false* to x_2. Given a PL formula and an interpretation, the truth value of a formula can be computed by a truth table or by induction. Considering the possible evaluations of a propositional variable x (i.e., *tt* or *ff*), we can construct the truth table for the logical operators $\neg, \wedge, \vee, \Rightarrow$, $\Leftrightarrow$ and $\oplus$ as shown in Table 2.1. It is important to note that $x_1 \Rightarrow x_2$ is false *iff* x_1 is true and x_2 is false.

x_1	x_2	$\neg x_1$	$x_1 \wedge x_2$	$x_1 \vee x_2$	$x_1 \Rightarrow x_2$	$x_1 \Leftrightarrow x_2$	$x_1 \oplus x_2$
ff	*ff*	*tt*	*ff*	*ff*	*tt*	*tt*	*ff*
ff	*tt*	*tt*	*ff*	*tt*	*tt*	*ff*	*tt*
tt	*ff*	*ff*	*ff*	*tt*	*ff*	*ff*	*tt*
tt	*tt*	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*

Table 2.1: Truth table.

We also describe an inductive definition of PL's semantics that defines the meaning of basic operators and also the meaning of more complex formulae in terms of the basic operators. We write $I \models \phi$ if ϕ evaluates to *tt* under I and $I \not\models \phi$ if ϕ evaluates to *ff* under I.

Definition 2.5. *We define the evaluation of formula* ϕ *under an interpretation* I *as follows.*

- $I \models x$ *iff* $I[x] = tt$

- $I \models \neg\phi$ *iff* $I \not\models \phi$
- $I \models \phi_1 \wedge \phi_2$ *iff* $I \models \phi_1$ *and* $I \models \phi_2$

Lemma 2.6. *The semantics of more complex formulae are evaluated as:*

$I \models \phi_1 \vee \phi_2$ *iff* $I \models \phi_1$ *or* $I \models \phi_2$

$I \models \phi_1 \Rightarrow \phi_2$ *iff, whenever* $I \models \phi_1$ *then* $I \models \phi_2$

$I \models \phi_1 \Leftrightarrow \phi_2$ *iff* $I \models \phi_1$ *and* $I \models \phi_2$*, or* $I \not\models \phi_1$ *and* $I \not\models \phi_2$

As an example, consider the formula $\phi : x_1 \vee x_2 \Rightarrow x_1 \wedge x_2$ under the interpretation $I : \{x_1 \mapsto \mathit{ff}, x_2 \mapsto \mathit{tt}\}$. We can compute the truth value of ϕ as follows

1. $I \not\models x_1$ since $I[x_1] = \mathit{ff}$
2. $I \models x_2$ since $I[x_2] = \mathit{tt}$
3. $I \models x_1 \vee x_2$ by 2 and semantics of the operator $\vee$
4. $I \not\models x_1 \wedge x_2$ by 1 and semantics of the operator $\wedge$
5. $I \not\models \phi$ by 3 and 4 and semantics of the operator $\Rightarrow$

2.1.2 Decision Procedures for Satisfiability

Section 2.1.1 introduced the truth table and semantic argument methods for determining the *satisfiability* of PL formulae. However, an algorithmic method can easily be implemented in order to decide *satisfiability* of PL formulae.

Definition 2.7. *A PL formula is satisfiable with respect to a class of interpretations if there exists an assignment to its variables under which the formula evaluates to true.*

The input of the algorithm to check the satisfiability is usually a PL formula in conjunctive normal form (CNF).

Definition 2.8. *Formally, a PL formula ϕ is in conjunctive normal form if it consists of a conjunction of one or more clauses, where each clause is a disjunction of one or more literals. It has the form $\bigwedge_i \left(\bigvee_j l_{ij}\right)$, where each l_{ij} is a literal.*

A PL formula can easily be transformed into an equisatisfiable CNF formula in polynomial time using Tseitin's encoding [172].

Definition 2.9. *Two PL formulae are said to be equisatisfiable if they are both satisfiable or they are both unsatisfiable.*

In Tseitin's encoding, we add a new literal to each logical operator (e.g., $\wedge$, $\vee$, and $\neg$) in the original PL formula, and several clauses to constrain the value of this literal to be equal to the expression it represents. The original PL formula is satisfiable *iff* the conjunction of these clauses together with the new literal is satisfiable.

As an example, consider the following PL formula:

$$x_1 \Rightarrow (\neg x_2 \vee x3) \tag{2.1}$$

For this example, let us assign the variable b_3 to the subexpression $\neg x_2$, b_2 to the subexpression $b_3 \vee x_3$, and b_1 to the implication $x_1 \Rightarrow b_2$, which is also the topmost operator of this formula. We need to satisfy b_1, together with three equivalences, as follows:

$$\begin{aligned} & b_1 \Leftrightarrow x_1 \Rightarrow b_2 \\ & b_2 \Leftrightarrow b_3 \vee x_3 \\ & b_3 \Leftrightarrow \neg x_2 \end{aligned} \tag{2.2}$$

The equivalences can be rewritten to CNF using Definition 2.2 as follows:

$$(\neg b_1 \vee \neg x_1 \vee b_2) \wedge (x_1 \vee b1) \wedge (b_1 \vee \neg b_2) \tag{2.3}$$

$$(\neg b_2 \vee b_3 \vee x_3) \wedge (\neg b_3 \vee b_2) \wedge (\neg x_3 \vee b_2) \tag{2.4}$$

$$(\neg b_3 \vee \neg x_2) \wedge (\neg x_2 \vee b_3) \tag{2.5}$$

The overall CNF formula is thus the conjunction of (2.3), (2.4), (2.5), and the unit clause (i.e., a clause that is composed of a single literal) b_1, which represents the topmost operator. The propositional satisfiability (SAT) problem is then to decide if there exists a satisfying assignment to the literals of the PL formula ϕ (in CNF) to satisfy all clauses. The algorithm to check the satisfiability of ϕ is a *decision procedure*, because given any formula, the algorithm always terminates with a "correct" yes/no answer after some finite amount of computation.

Modern decision procedures to check the satisfiability of PL formulae in CNF are based on a variant of the Davis-Putnam-Logemann-Loveland algorithm (DPLL), which consists essentially of two steps: *(i)* choose a truth value for some literal and *(ii)* propagate the implications of this decision that are easy to infer. This method is known as *unit propagation*, which can simplify a set of clauses and thus avoids a large part of the

naive search space. The algorithm backtracks when a *conflict* is reached and learns the assignments to literals of the conflict to avoid reaching the same conflict again.

In this context, a SAT solver is thus an algorithm (based on a variant of the DPLL) that takes as input a formula ϕ (which is in CNF) and decides whether it is *satisfiable* or *unsatisfiable*. The formula ϕ is said to be satisfiable (or sat) if the SAT solver is able to find an interpretation that makes the formula *true* (cf. Definition 2.7). The formula ϕ is said to be unsatisfiable (or unsat) if none of the interpretations make the formula *true*. In the satisfiable case, SAT solvers can provide a model, i.e., a satisfying assignment to the propositional variables of the formula ϕ. In the unsatisfiable case, when a SAT solver concludes that there is no satisfying assignment to ϕ, its internal steps for concluding this can be used to construct a *resolution proof* [154] (and most state-of-the-art SAT solvers can output such steps that can be used as an independently checkable proof of unsatisfiability).

Definition 2.10. *A resolution proof is a sequence of deduction steps based on the inference rule:*

$$\frac{p_1 \vee ... \vee p_n \vee (\alpha) \quad q_1 \vee ... \vee q_m \vee (\neg\alpha)}{p_1 \vee ... \vee p_n \vee q_1 \vee ... \vee q_m}$$

where $p_1 \vee \ldots \vee p_n, \vee q_1 \vee \ldots \vee q_m$ are literals and α is a variable (also called resolution variable). The clauses $p_1 \vee \ldots \vee p_n \vee (\alpha)$ and $q_1 \vee \ldots \vee q_m \vee (\neg\alpha)$ are called resolving and $p_1 \vee \ldots \vee p_n \vee q_1 \vee \ldots \vee q_m$ is called resolvent.

The intuitive interpretation of resolution is that to satisfy clauses $p_1 \vee \ldots \vee p_n \vee (\alpha)$ and $q_1 \vee \ldots \vee q_m \vee (\neg\alpha)$ that share the resolution variable α but disagree on its value, either the rest of $p_1 \vee \ldots \vee p_n$ or the rest of $q_1 \vee \ldots \vee q_m$ must be satisfied. For example, consider the following formula in CNF:

$$\phi : (p \vee \neg q) \wedge q \wedge \neg p \tag{2.6}$$

from resolution

$$\frac{p \vee \neg q \quad q}{p} \tag{2.7}$$

we can construct

$$\phi_1 : (p \vee \neg q) \wedge q \wedge \neg p \wedge p \tag{2.8}$$

from resolution

$$\frac{\neg p \quad p}{\square} \tag{2.9}$$

we can conclude that the original formula ϕ is unsatisfiable because the last deduction step (2.9) ends with empty clause $\square$. Therefore, we can also say that a PL formula in CNF is unsatisfiable *iff* there exists a finite series of deduction steps (based on the inference rule defined in (2.10)) ending with the empty clause.

Although PL formulae can be converted into CNF in polynomial time (using Tseitin's encoding as described above), the problem to decide satisfiability of PL formulae belongs to the well-known $\mathcal{NP}$-complete [55] class. Much research in the past decade has advanced the state-of-the-art considerably. For a recent survey on SAT we refer the reader to [24].

From the verification point of view, a propositional encoding and use of a SAT solver to reason about programs have two main limitations as follows. First, the size of the propositional encoding depends directly on the size of the basic data types and arrays occurring in the program. Consequently, large data-paths in programs involving complex expressions lead to large propositional formulae. Second, high-level information is lost when verification conditions are converted into propositional logic. SAT solvers operate at the bit-level and are thus unable to exploit the structure provided by the higher abstraction levels. These limitations can be substantially reduced by encoding word-level information in theories richer than propositional logic and using SMT solvers for the generated verification conditions.

2.1.3 Satisfiability Modulo Theories

SMT decides the satisfiability of certain first-order formulae using a combination of different background theories and thus generalizes propositional satisfiability by supporting uninterpreted functions, linear and non-linear arithmetic, bit-vectors, tuples, arrays, and other decidable first-order theories (FOL is in general undecidable [37]). Table 2.2 shows some examples of the decidable first-order theories (e.g., equality, bit-vectors, linear arithmetic, arrays) supported by typical SMT solvers.

Theory	**Example**
Equality	$z_1 = z_2 \wedge \neg(z_1 = z_3) \Rightarrow \neg(z_2 = z_3)$
Bit-vectors	$((b >> i) \| 2) \& 1 = 1$
Linear Arithmetic	$(4y_1 + 3y_2 + 1 \geq 4) \vee (y_2 - 3y_3 + 5 \leq 3)$
Arrays	$(j = k \wedge select(a,k) = 2) \Rightarrow select(a,j) = 2$
Combined Theories	$g(select(store(a,c,12),c)) \neq g(1) \wedge c - 3 = c - 3$

Table 2.2: Examples of First-Order Theories.

A first-order *theory* $\mathcal{T}$ is defined by a *signature* Σ that consists of a set of functions,

predicates, and constant symbols (also called nonlogical symbols) and a set of *axioms* A that consists of first-order logic formulae in which the only nonlogical symbols that appear are in Σ [29]. A **Σ-formula** is a formula that uses nonlogical symbols of Σ, as well as variables, logical connectives ($\wedge$, $\vee$, $\neg$), quantifiers ($\exists$ and $\forall$) and parentheses.

Definition 2.11. *Given a Σ-theory $\mathcal{T}$ and a quantifier-free formula ψ, we say that ψ is $\mathcal{T}$-satisfiable if and only if there exists a structure that satisfies both the formula and the sentences of $\mathcal{T}$, or equivalently, whether $\mathcal{T} \cup \{\psi\}$ is satisfiable.*

Definition 2.12. *Given a set $\Gamma \cup \{\psi\}$ of first-order formulae over a Σ-theory $\mathcal{T}$, we say that ψ is a $\mathcal{T}$-consequence of Γ, and write $\Gamma \models_{\mathcal{T}} \psi$, if and only if every model of $\mathcal{T} \cup \Gamma$ is also a model of ψ. Checking $\Gamma \models_{\mathcal{T}} \psi$ can be reduced in the usual way to checking the $\mathcal{T}$-satisfiability of $\Gamma \cup \{\neg\psi\}$.*

State-of-the-art SMT solvers are built on top of efficient SAT solvers to speed up the performance and support the combination of different decidable theories [20, 31, 57]. For example, SAT solvers do not scale well when reasoning on the propositional encoding of arithmetical operators (e.g., multiplication), because the operands are treated as arrays of Booleans and most of the computational effort might be wasted during the boolean search (e.g., up to 2^w factor in the amount of boolean search, where w represents the width of the data type) [27]. SMT solvers, however, often integrate a simplifier, which applies standard algebraic reduction rules (e.g., $r \wedge \mathit{false} \mapsto \mathit{false}$) and contextual simplification (e.g., $a = 7 \wedge p(a) \mapsto a = 7 \wedge p(7)$) before replacing the word-level operators by bit-level circuit equivalents (i.e., before bit-blasting). Furthermore, SMT solvers (e.g., [32]) often implement an incremental and layered approach which permits strengthening incrementally the model of the arithmetic operators and they thus achieve performance improvements of several orders of magnitude when compared to plain bit-blasting, as reported in [28]. Consequently, as structural word level information (i.e., predicates from various decidable theories) remains in the problem formulation, then *bit-blasting* is used by the SMT solvers only as a last resort if higher level and less expensive techniques are not enough to solve the problem at hand.

The SMT-LIB initiative [164] aims at establishing a common standard for the specification of background theories, but the background theories still vary and most of current SMT solvers provide functions in addition to those specified in the SMT-LIB. Therefore, we describe here all the fragments that we found in the SMT solvers CVC3 [20], Boolector [31] and Z3 [57] for the theory of linear, non-linear, and bit-vector arithmetic. We summarize the syntax of these background theories as follows:

Note that here we use standard notation to describe the above grammar, and we thus only focus on certain aspects of the notation. In this grammar *Fml* denotes Boolean-valued expressions, *Trm* denotes terms built over integers, reals, and bit-vectors while *op* denotes binary operators. The logical connectives *con* consist of conjunction ($\wedge$), disjunction ($\vee$), exclusive-or ($\oplus$), implication ($\Rightarrow$), and equivalence ($\Leftrightarrow$). The interpretation

$$
\begin{array}{rcl}
Fml & ::= & Fml\ con\ Fml \mid \neg Fml \mid Atom \\
con & ::= & \wedge \mid \vee \mid \oplus \mid \Rightarrow \mid \Leftrightarrow \\
Atom & ::= & Trm\ rel\ Trm \mid Var \mid true \mid false \\
rel & ::= & < \mid \leq \mid > \mid \geq \mid = \mid \neq \\
Trm & ::= & Trm\ op\ Trm \mid \sim Trm \mid Var \mid Const \\
 & & \mid select(Trm, i) \mid store(Trm, i, v) \\
 & & \mid Extract(Trm, i, j) \mid SignExt(Trm, k) \mid ZeroExt(Trm, k) \\
 & & \mid ite(Fml,\ Trm,\ Trm) \\
op & ::= & + \mid - \mid * \mid / \mid rem \mid \texttt{<<} \mid \texttt{>>} \mid \& \mid | \mid \oplus \mid @
\end{array}
$$

FIGURE 2.1: Syntax of the Background Theories

of the relational operators (i.e., $<$, $\leq$, $>$, $\geq$) and the non-linear arithmetic operators (i.e., $*$, $/$, *rem*) depend on whether their arguments are unsigned or signed bit-vectors, integers or real numbers. Here, the operator *rem* denotes the signed or unsigned remainder, depending on the arguments. The left- and right-shift operators (i.e., `<<`, `>>`) depend on whether an unsigned or signed bit-vectors encoding is used. We assume that the type of the expression is clear from the context. The bit-wise operators consist of and (&), or (|), exclusive-or ($\oplus$), complement ($\sim$), right-shift (`>>`), and left-shift (`<<`). $Extract\,(Trm, i, j)$ denotes bit-vector extraction from bits i down to j to yield a new bit-vector of size $i - j + 1$ while the operator @ denotes the concatenation of the given bit-vectors. $SignExt\,(Trm, k)$ extends the bit-vector to the signed equivalent bit-vector of size $w + k$, where w is the original width of the bit-vector, while $ZeroExt\,(Trm, k)$ extends the bit-vector with zeros to the unsigned equivalent bit-vector of size $w + k$. The conditional expression $ite(f, t_1, t_2)$ takes as its first argument a Boolean formula f and depending on its value selects either the second or the third argument.

The array theories of SMT solvers are typically based on the two McCarthy axioms [123]. The function *select(a, i)* denotes the value of array a at index position i and *store(a, i, v)* denotes an array that is exactly the same as array a except that the value at index position i is v if i is within the array bounds and unspecified otherwise. Formally, the functions *select(a, i)* and *store(a, i, v)* can then be characterized by the following two axioms [20, 31, 57]:

$$
\begin{array}{l}
i = j \Rightarrow select\,(store\,(a, i, v)\,, j) = v \\
i \neq j \Rightarrow select\,(store\,(a, i, v)\,, j) = select\,(a, j)
\end{array}
$$

The first axiom asserts that the value selected at index j is the same as the last value stored to the index i if the two indices i and j are equal. The second axiom asserts that storing a value to index i does not change the value at index j, if the indices i and j are different. In addition to that, equality on array elements is defined by the theory of equality with uninterpreted functions (i.e., $a = b \wedge i = j \Rightarrow select\,(a, i) = select\,(b, j)$) and the extensional theory of arrays then allows reasoning about array comparisons as

follows [20, 31, 57]:

$$a = b \Leftarrow \forall i \cdot select\,(a, i) = select\,(b, i)$$
$$a \neq b \Rightarrow \exists i \cdot select\,(a, i) \neq select\,(b, i)$$

The theory of arrays employs the notion of unbounded arrays size, but arrays in software are typically of bounded size. This means that if an index variable i exceeds the size of an array in a program, the value returned might be undefined or a crash might occur. Chapter 3 shows how to generate verification conditions in order to check for array bounds violation in programs.

Another theory of interest to software verification is the theory of tuples, where it allows us to model the ANSI-C *struct* and *union* datatypes. They provide store and select operations similar to those in arrays, but working on the tuple elements. Each field of the tuple is represented by an integer number. Hence, the expression *select(t, f)* denotes the field f of tuple t while the expression *store(t, f, v)* denotes a tuple t that at field f has the value v and all other tuple elements remain the same. Chapter 3 shows how structures and unions are encoded using the theory of tuples.

As a running example for background theories, we give a simple SMT formula that uses three theories (bit-vector arithmetic, theory of arrays, and uninterpreted functions). Let a be an array, b, c and d be signed bit-vectors of width 16, 32 and 32 respectively, and let g be an unary function. The function g implies that for all x and y (where x and y are variables), if $x = y$, then $g\,(x) = g\,(y)$ (congruence rule). Formally, the unary function g instantiates to the following axiom: $\forall x, y.x = y \Rightarrow g\,(x) = g\,(y)$. In other words, we say that function g always produces the same result when applied to the same arguments [29, 58, 133].

$$g\,(select\,(store\,(a, c, 12)\,, SignExt\,(b, 16) + 3)) \neq g\,(SignExt\,(b, 16) - c + 4)$$
$$\wedge SignExt\,(b, 16) = c - 3 \wedge c + 1 = d - 4$$

In order to sum $SignExt\,(b, 16){+}3$, subtract $SignExt\,(b, 16){-}c$ and compare $SignExt\,(b, 16) = c - 3$, we have first to expand the term $SignExt\,(b, 16)$ so that the resulting bit-vector, say b', extends b to the signed equivalent bit-vector of size 32 (i.e., $SignExt\,(b, 16)$ thus extends b to the size $w + 16$, where w is the original width of the bit-vector b). After expanding the term $SignExt\,(b, 16)$, we then obtain the following formula:

$$g\,(select\,(store\,(a, c, 12)\,, b' + 3)) \neq g\,(b' - c + 4) \wedge b' = c - 3 \wedge c + 1 = d - 4$$

Now the bit-vectors b' and c have the same width. One way of checking the satisfiability of this formula is to replace b' by $c - 3$ in the inequality so that we obtain an equivalence formula such as:

$$g\left(select\left(store\left(a,c,12\right),c-3+3\right)\right) \neq g\left(c-3-c+4\right) \wedge c-3=c-3 \wedge c+1=d-4$$

after using facts about bit-vector arithmetic, this formula can be rewritten as:

$$g\left(select\left(store\left(a,c,12\right),c\right)\right) \neq g\left(1\right) \wedge c-3=c-3 \wedge c+1=d-4$$

Finally, the theory of arrays implies that the select/store functions reduce the arguments of function $g\left(select\left(store\left(a,c,12\right),c\right)\right)$ to $g\left(12\right)$ and the formula becomes:

$$g\left(12\right) \neq g\left(1\right) \wedge c-3=c-3 \wedge c+1=d-4$$

Consequently, the formula above is satisfiable since there is an assignment to the bit-vectors c (e.g., $c = 5$) and d (e.g., $d = 10$) such that the first ($g\left(12\right) \neq g\left(1\right)$), second ($c-3=c-3$) and third ($c+1=d-4$) terms hold.

2.1.4 Linear-time Temporal logic

Linear-time temporal logic, or simply LTL, is a commonly used specification logic in bounded model checking [22, 94, 101], which extends propositional logic (discussed in Subsection 2.1.1) by including temporal operators. It models time by means of a sequence of states (denoted by $s_i \in S$, where i indicates a state in a given time step and S is the set of states), or computation path (henceforth called π), extending infinitely into the future (hence the term "linear", which means that at each state in time there is a single successor state). In LTL, we are thus able to specify properties of the type "for some state on the path" or "for every two consecutive states".

Definition 2.13. *The syntax of LTL is defined over a set of atomic propositions, logical operators and temporal operators as follows:*

$$\begin{array}{rcl}\phi & ::= & \top \mid \bot \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \\ & & \mid X\phi \mid F\phi \mid G\phi \mid A\phi \mid \phi_1\, U \phi_2 \mid \phi_1 R\, \phi_2\end{array}$$

The symbols $\top$ and $\bot$ are atoms and represent *true* and *false* respectively (as described in Subsection 2.1.1). The logical operators include *negation* ($\neg$), *conjunction* ($\wedge$), *disjunction* ($\vee$) and *implication* ($\Rightarrow$). The temporal operators are "next state" (X), "some future state (eventually)" (F), "all future states (globally)" (G), "along all computation paths" (A), "until" (U) and "release" (R). An LTL formula can be evaluated over a computation path π (i.e., $\pi = s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$) or over a set of states. LTL formulae are thus of two kinds: computation path and state formulae. The intuitive interpretation of the operators X, G, F, U and R over computation path formulae is as follows:

- $X\ \phi$ means that ϕ has to hold at the ne**X**t point in time.
- $F\ \phi$ means that ϕ has to hold at some point in the **F**uture.
- $G\ \phi$ means that ϕ has to hold **G**lobally (at all future points).
- $\psi\ U\ \phi$ means that ψ has to hold continuously **U**ntil ϕ holds.
- $\psi\ R\ \phi$ means that ϕ has to remain *true* up to and including the moment when ψ first becomes *true*; if ψ never becomes *true*, ϕ must remain *true* forever; ψ **R**eleases ϕ.

and the interpretation of the operator A over state formulae is as follows:

- $A\ \phi$ means that ϕ has to hold along **A**ll computation paths.

The operators X, F, G and A are unary, so that $X\ \phi$, $F\ \phi$, $G\ \phi$ and $A\ \phi$ are well-formed formula whenever ϕ is a well-formed formula. The operators U and R are binary, so that $\psi\ U\ \phi$ and $\psi\ R\ \phi$ are well-formed formula whenever both ψ and ϕ are well-formed formulae. We omit the W operator because R and W are actually quite similar; the differences are that they swap the roles of ψ and ϕ, and the clause for W has an $i-1$ where R has i (see below the satisfaction relation of the LTL formulae). Figure 2.2 shows the informal semantics of the LTL operators so that each operator is shown in a computation path π, where each dot represents a state in time (e.g., $s_1, s_2, s_3, \ldots$).

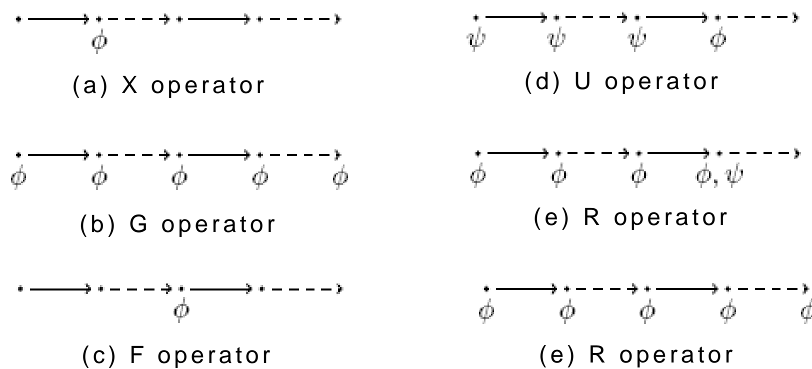

FIGURE 2.2: LTL semantics for the operators X, G, F, U, and R (when ψ first becomes *true* and when ψ never becomes *true*) over π [94].

Software systems are typically modelled by means of a state transition system M (also called *model*).

Definition 2.14. *A state transition system, denoted by M, is defined by a triple (S, R, S_0) where S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $S_0 \subseteq S$ represents the set of initial states.*

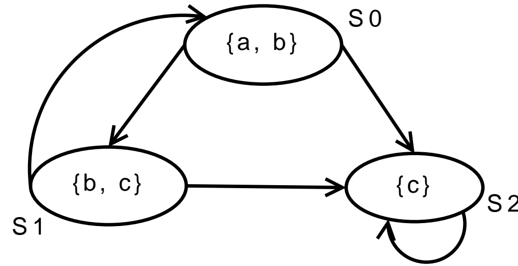

FIGURE 2.3: Example of a Kripke structure (with deadlock) for states s_0, s_1, and s_2 (where s_2 has a transition back to itself).

In software systems, a state represents the assignment of values (e.g., Booleans, integers, characters) to variables. The semantics of an LTL formula is then defined along a computation path $\pi = s_1 \to s_2 \to \ldots \to s_n$, which is a sequence of states over M. A program thus defines the form of its states, the set of transitions between the states, and the set of computations that it can potentially produce. The set of computations of a program defines the program itself with the same precision of its source code.

Formally, let π^i be a computation path π with a designated formula evaluation position i. We assume a labelling (or interpretation) function $L : S \Rightarrow 2^P$ mapping L from each state to the set of propositional variables represented by P. For example, the power set of $\{a, b\}$ is $\{\oslash, \{a\}, \{b\}, \{a, b\}\}$ and L is just an assignment of truth values to all the propositional variables, exactly as it was for the case of *interpretation* of PL formulae. To help us define the semantics of LTL formulae, we then extend definition 2.14 to include the labelling function $L : S \Rightarrow 2^P$ so that M now becomes a quadruple $K = (S, R, S_0, L)$, which is called a Kripke structure.

Definition 2.15. A Kripke structure is a quadruple $K = (S, R, S_0, L)$ consisting of a set of states S, a set of transitions R, a set of initial states S_0 (as defined in 2.14) and a labelling function $L : S \Rightarrow 2^P$, which defines for each state $s \in S$ the set $L(s)$ of all propositional variables that belong to s.

Note that we can construct an infinite path in a Kripke structure and thus a *deadlock* state (i.e., a state with a transition back to itself) might occur in K. Figure 2.3 shows an example of representation of K, which consists of three states s_0, s_1 and s_2 with transitions $s_0 \to s_1$, $s_0 \to s_2$, $s_1 \to s_0$, $s_1 \to s_2$ and $s_2 \to s_2$; and $L(s_0) = \{a, b\}$, $L(s_1) = \{b, c\}$ and $L(s_2) = \{c\}$.

Definition 2.16. *Let $K = (S, R, S_0, L)$ be a model of our system and π be a path in K. The formal semantics whether π satisfies an LTL formula ϕ is thus defined by the satisfaction relation $\pi \models \phi$, which extends the satisfaction relation of PL formulae over temporal operators, as follows:*

$$
\begin{array}{lll}
\pi^i \models p & \textit{iff} & p \in L(s_i) \\
\pi^i \models \neg p & \textit{iff} & \pi^i \not\models p \\
\pi^i \models \phi_1 \wedge \phi_2 & \textit{iff} & \pi^i \models \phi_1 \textit{ and } \pi^i \models \phi_2 \\
\pi^i \models \phi_1 \vee \phi_2 & \textit{iff} & \pi^i \models \phi_1 \textit{ or } \pi^i \models \phi_2 \\
\pi^i \models X\,\phi & \textit{iff} & \pi^{i+1} \models \phi \\
\pi^i \models F\,\phi & \textit{iff} & \textit{for some } i \geq 1 \textit{ such that } \pi^i \models \phi \\
\pi^i \models G\,\phi & \textit{iff} & \textit{for all } i \geq 1, \pi^i \models \phi \\
\pi^i \models \phi_1\ U\ \phi_2 & \textit{iff} & \exists j \geq i \textit{ such that } \pi^j \models \phi_2 \textit{ and } \pi^n \models \phi_1 \textit{ for all } i \leq n < j \\
 & & \textit{we have } \pi^j \models \phi_1; \textit{ or for all } k \geq 1 \textit{ we have } \pi^k \models \phi_1 \\
\pi^i \models \phi_1 R\, \phi_2 & \textit{iff} & \textit{for all } j \geq i : \pi^j \models \phi_2 \textit{ or } \pi^n \models \phi_1 \textit{ for some } i \leq n < j \\
\pi^i \models A\,\phi & \textit{iff} & \pi \models \phi \textit{ for all paths } \pi \textit{ starting in } s_i
\end{array}
$$

According to definition 2.16, the following LTL formulae hold in the transition system of Figure 2.3:

- $s_0 \models A\,(a \wedge b)$
- $s_1 \models A\ (b\,U\,c)$
- $s_2 \models A\,G\,c$

and the following LTL formulae do not hold in the transition system of Figure 2.3:

- $s_0 \not\models A\,X\,(b \wedge c)$
- $s_1 \not\models A\,G\,c$
- $s_2 \not\models A\,G\,F\,a$

As an example of how LTL is used to specify properties, consider the classic mutual exclusion problem in which two threads, say T_1 and T_2, cannot have simultaneous access to a common resource CR. Thread T_i is essentially modelled by three locations as follows:

1. the *noncritical* section (i.e., a section that does not need exclusive access to CR);
2. the *waiting* phase, which is entered when the thread intends to enter the critical section, i.e., access CR; and
3. the *critical* section (i.e., a section that accesses CR that must not be concurrently accessed by more than one thread).

Let the propositions c_1 and c_2 denote that thread T_1 and T_2 are in their critical section. The safety property stating that T_1 and T_2 never simultaneously have access to their critical sections (i.e., at most one thread is in critical section at any time) can be described by the following LTL formula:

$$AG\,(\neg c_1 \vee \neg c_2) \tag{2.10}$$

This formula expresses that for all paths π at least one of the two threads is not in its critical section (expressed by $\neg c_i$).

2.2 Bounded Model Checking of Software

This section presents the formulation of the BMC technique, an overview of completeness methods to prove properties in the BMC framework, and describes typical BMC architectures used in software verification. It also compares the BMC technique to other state-of-the-art software verification techniques that are currently used in practice.

2.2.1 Formulation

Bounded model checking (BMC) has been successfully applied to verify software systems and discovered subtle errors in commercial products. The idea of BMC is to unwind the program and the correctness properties k times, and generate a propositional formula that is satisfiable if and only if a counterexample of size k (or smaller) exists [25]. However, the technique is not complete because there might still be a counterexample that is longer than k. Completeness can only be ensured if we know an upper bound on the depth of the state space, i.e., if we can ensure that we have already explored all the relevant behaviour of the system, and searching any deeper only exhibits states that have already been checked. In BMC of software, the bound k limits the number of loop iterations and recursive calls occurring in the program. BMC thus analyzes only bounded program runs and thereby achieves decidability since software verification in general is undecidable due to infinite program runs (e.g., in reactive or interactive software systems).

Formally, given a temporal logic property ϕ to be verified on a finite transition system M (cf. Definition 2.14), BMC unwinds the system k times and translates it into a verification condition ψ such that ψ is satisfiable if and only if ϕ has a counterexample (i.e., a behaviour which falsifies the property ϕ) of depth less than or equal to k. The propositional problem associated with SAT-based BMC is formulated by constructing the following formula [25]:

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg\phi_k \tag{2.11}$$

Here, ϕ_k represents a safety property ϕ in step k, I is the set of initial states of M, $R(s_i, s_{i+1})$ is the transition relation of M at time steps i and $i+1$. Hence, the formula $\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ represents the set of all executions of M of length k. $\neg\phi_k$ represents the condition that ϕ is violated in state k, which is reached by a bounded execution of M of length k. Finally, the resulting (bit-vector) formula is translated to conjunctive normal form in linear time and passed to a SAT solver for checking satisfiability. Formula (2.11) can be used to check safety properties [149]. Liveness properties (e.g., starvation, deadlock) that contain the LTL operator F are checked by encoding $\neg\phi_k$ in a loop within a bounded execution of length at most k, such that ϕ is violated on each state in the loop. In this case, formula 2.11 can be rewritten as:

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{k} \neg\phi_i\right) \tag{2.12}$$

where ϕ_i is the propositional variable ϕ at time step i. Thus, this formula can be satisfied if and only if for some i $(i \leq k)$ there exists a reachable state at time step i in which ϕ is violated.

Definition 2.17. *Let M be a transition system. A state $s \in S$ is called a reachable state in M if there exists a finite sequence of state transitions starting from an initial state s_0 and ending in state s, i.e., $s_0 \overset{R_0}{\rightarrow} s_1 \overset{R_1}{\rightarrow} \ldots \overset{R_n}{\rightarrow} s_n = s$, where $s_0 \overset{R_0}{\rightarrow} s_1$ denotes a state transition by applying R_0.*

However, in software verification, the more common application of BMC relies on checking safety properties that contain the LTL operator G. They are typically formalized using *assert* statements that encode the properties that have to hold at the respective location. The safety properties in single- and multi-threaded programs typically check for out-of-bounds array indexing, NULL-pointer dereferencing, memory leaks, data race, atomicity and order violations, and arithmetic overflow.

2.2.2 Verification Conditions

BMC analyzes only bounded program runs, but generates verification conditions (VCs) that reflect the exact path in which a statement is executed, the context in which a given function is called, and the bit-accurate representation of the expressions. A verification condition is a logical formula (constructed from the bounded program and desired correctness properties) whose validity implies that the program's behaviour agrees with its specification [12, 29, 74, 109]. Correctness properties in programs can be specified by the

user via *assert* statements or automatically generated from a specification language as in [16]. If all of a bounded program's VCs are valid, then the program is in compliance with its specification up to the given bound.

In this thesis, we are concerned with verification conditions expressed in quantifier-free first-order logic formulae (over finite data structures) such as those presented in Table 2.2. As example, we consider a simple C program (slightly modified from [109]) with an exponential number of paths as shown in Figure 2.4(a); the corresponding C program in single static assignment (SSA) form [9] is shown in Figure 2.4(b). Note that the SSA form is an intermediate representation used by compilers to facilitate optimizations and transformations of the program code. The common property in SSA form is that every variable has only one definition in the program text. This is achieved by introducing a fresh variable from the original name (e.g., with a subscript) such that every assignment has a unique left hand side as shown in Figure 2.4(b).

```
1  #include <assert.h>
2  int x[N], a;
3  ...
4  int main(void) {
5    a=N;
6    for(int i=0; i<N; i++)
7      if (x[i]>1)
8        a--;
9    assert(a<=N);
10   return 0;
11 }
```

(a)

```
1 a1 = N
2 a2 = (x[0] > 1) ? a1 - 1 : a1
3 a3 = (x[1] > 1) ? a2 - 1 : a2
4 a4 = (x[2] > 1) ? a3 - 1 : a3
5 ...
6 an+1 = (x[N-1] > 1) ? an - 1 : an
```

(b)

FIGURE 2.4: (a) A simple C program with a *for* loop. (b) The corresponding unwound C program of (a) converted into SSA form.

Apart from that, this program has an exponential number of paths since each element of array x can be either greater than one or less than or equal to one. Despite the large number of paths through the program, BMC unwinds it up to a bound k and translates it into a VC ψ such that ψ is satisfiable if and only if the assertion $(a <= N)$ fails. Note that BMC still encodes the states of the program with a size that grows linearly with N). More precisely, the program in Figure 2.4(a) is converted into the ψ using first-order logic as follows:

$$\psi := \begin{bmatrix} a_1 = N \\ \wedge\ a_2 = ite\left(x[0] > 1, a_1 - 1, a_1\right) \\ \wedge\ a_3 = ite\left(x[1] > 1, a_2 - 1, a_2\right) \\ \wedge\ a_4 = ite\left(x[2] > 1, a_3 - 1, a_3\right) \\ \wedge \ldots \\ \wedge\ a_{N+1} = ite\left(x[N-1] > 1, a_N - 1, a_N\right) \\ \wedge \neg\left(a_{N+1} \leq N\right) \end{bmatrix} \tag{2.13}$$

The ternary operator $f\ ?\ t_1 : t_2$ shown in Figure 2.4(b) is converted into the conditional expression $ite(f, t_1, t_2)$ that takes as its first argument the Boolean formula f and depending on its value selects either the second (i.e., t_1) or the third argument (i.e., t_2). In order to verify that the assertion $(a <= N)$ holds, its negation is added to ψ and we check whether the entire formula is satisfiable using an off-the-self SMT solver. As described in Section 2.1, Formula (2.13) can simply be represented as a Boolean logic circuit, which can further be transformed into a (equisatisfiable) CNF formula over propositional variables by Tseitin's transform [172] in linear time and by introducing at most a linear number of fresh variables. However, checking the validity of a first-order logic formula in a given background theory is an $\mathcal{NP}$-complete problem [145]).

2.2.3 Completeness

Bounded model checking can be used to find property violations up to the bound k but not to prove properties, unless an upper bound is known on the depth of the state space, which is not generally the case. For software verification, we can adopt two different strategies in order to *prove* properties: *(i)* compute the *completeness threshold*, which can be smaller than or equal to the maximum number of loop-iterations occurring in the program or *(ii)* determine the high-level worst-case execution time (WCET), which also gives a bound on the maximum number of loop-iterations [24, 45, 72]. However, in practice, complex software systems involve large data-paths and complex expressions. Therefore, the verification conditions that arise from BMC of programs become harder to solve and require substantial amounts of memory to build.

2.2.3.1 Craig Interpolation

One feasible alternative to prove properties in BMC is to compute the Craig interpolants for inconsistent pairs (or more generally, sets) of formulae [125, 126, 127, 128]. This alternative approach exploits the SAT/SMT solvers' ability to produce refutations, i.e., proofs that there is no counter-example of depth less than or equal to k. This proof does not ensure whether a given property holds in the model, but it contains information about the reachable states of the model.

Definition 2.18. *Given a pair of formulae (A, B), and a proof by resolution for (A, B), an interpolant for (A, B) is a formula F with the following properties [125, 126]:*

- $A \Rightarrow F$
- $F \wedge B$ *is unsatisfiable*
- F *refers only to the common variables of A and B*

As an example, consider $A = (x_1 \wedge x_2)$ and $B = (\neg x_2 \wedge x_3)$. Given that $(x_1 \wedge x_2)$ must imply F (or simply that $\neg x_1 \vee \neg x_2 \vee F$ hold) and $F \wedge \neg x_2 \wedge x_3$ must be unsatisfiable, one possible interpolant for the given pair of formulae (A, B) is $F = x_2$ since x_2 is also common to both A and B.

The use of interpolants allows us to define a complete method for finite-state reachability analysis based on SAT and SMT solvers. In order to show how BMC and interpolation can be combined, we refer to Section 2.2.1 where we define the Formula (2.11) and the terms I, R, and ϕ. Now suppose that $Q = I$ and we partition Formula (2.11) so that the set of initial states I and the first instance of the transition relation R are in set A, while the remaining instances of R and the property ϕ are in set B as shown in Figure 2.5 (note that k is unknown).

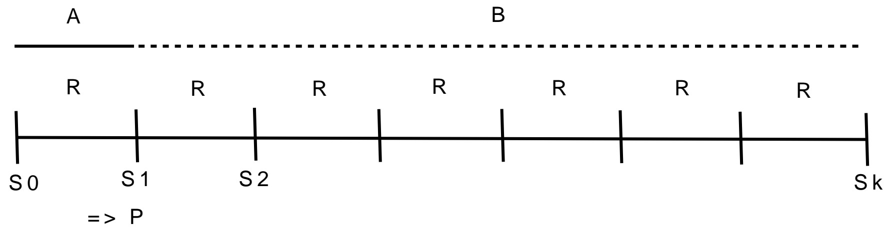

FIGURE 2.5: Computing image by interpolation [125].

Suppose that we use an SMT solver to prove that the $A \wedge B$ is unsatisfiable, i.e., we use an SMT solver to conclude that there is no satisfying assignment to $A \wedge B$.[1] The internal steps performed by the SMT solvers for reaching this conclusion can be used to construct a proof of unsatisfiability Π. From this proof, we can derive an interpolant F for the pair of formulae (A, B), i.e., $F = interpolant\,(\Pi, A, B)$. According to Definition 2.18, A must imply F and since we defined A to be the set of initial states and the first instance of R (i.e., from Figure 2.5, $A = s_0 \wedge s_1$), it follows that F is *true* in every state reachable from the initial state in one step. In other words, we can say that F is an over-approximation of the forward image of I [125, 126]. Also according to Definition 2.18, the formula $F \wedge B$ must be unsatisfiable (from Figure 2.5, $B = s_2 \wedge s_3 \wedge \ldots \wedge s_k$), which means that there is no state satisfying F that can reach a final state s_k. After computing the interpolant

[1]Note that if at any stage we can satisfy the property ϕ within k steps from the initial state, then we have found a counterexample.

F, we then check whether F implies Q. If F implies Q, then no reachable state can satisfy the property ϕ and we can thus conclude that the property holds. However, since F is an approximation, we can falsely conclude that the final state is reachable. In this case, we update $Q = F \vee Q$ and $A = F \wedge R_0$, increase the value of $k+1$ and check whether $A \wedge B$ is unsatisfiable. If $A \wedge B$ is satisfiable, we have found a valid counter-example (i.e., a path from the initial state to the final state). Otherwise, we compute the interpolant $F = interpolant\,(\Pi, A, B)$ again and check whether F implies Q. We stop this procedure when we have found a valid counter-example or have proved that the final state is not reachable (i.e., the property holds). The details of the algorithm and further information about the use of interpolants in model checking can be found in [125, 126, 127, 128].

2.2.3.2 K-Induction

Another feasible alternative to prove properties in BMC is to compute invariants by means of induction [162, 66]. The k-induction method has been successfully applied to verify hardware designs (represented as finite state machines) using a SAT solver, but the first attempts to apply this technique to software are only very recent [63]. In order to present the k-induction method, we use the notation of [63, 66], which describes the principle via temporal induction (i.e., the induction is carried out over the time steps of the finite state machines). The simplest form of k-induction consists of two steps: the *base-case* and the *induction-step*. Let $I(s)$ and $R(s, s')$ encode the set of initial states and transition relation of the finite transition system M, and let $P(s)$ denote states satisfying a safety property ϕ (recall Definition 2.14). The strengthened induction, as proposed in [66], is then defined by the following formulae:

$$
\begin{aligned}
Base_k &= I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k) \wedge (\neg P(s_0) \vee \ldots \vee \neg P(s_k)) \\
Step_k &= P(s_1) \wedge R(s_1, s_2) \wedge \ldots \wedge P(s_k) \wedge R(s_k, s_{k+1}) \wedge \neg P(s_{k+1})
\end{aligned}
\tag{2.14}
$$

The intuitive interpretation of these two formulae are as follows: in the base-case, we aim to check that P holds in all states reachable from an initial state within k steps (we assume that $k \geq 0$) and in the induction-step, we aim to check that whenever P holds in k consecutive states $s_1, \ldots, s_k$, P also holds in the next state s_{k+1} of the system. In both cases, we check whether formulae $Base_k$ and $Step_k$, as described above, are unsatisfiable. An algorithm can then be devised from these two formulae, which unwinds the system design incrementally and check whether $Base_k$ is satisfiable or $Step_k$ is unsatisfiable in order to determine termination. In particular, if $Base_k$ turns to be satisfiable in time step k, then we have found a violation of the property. If $Step_k$ is unsatisfiable in time step k, then the property holds.

2.2.4 BMC Architecture

Here, we overview typical BMC architectures used in software verification, focusing on the most prominent example, the C Bounded Model Checker (CBMC) [42, 41, 106]. The CBMC tool implements the BMC technique for ANSI-C/C++ programs using SAT solvers. CBMC can process C/C++ code using the goto-cc tool [179], which compiles the C/C++ code into equivalent GOTO-programs (i.e., control-flow graphs) using a gcc-compliant style. The GOTO-programs can then be processed by the symbolic execution engine. Alternatively, CBMC uses its own, internal parser based on Flex/Bison, to process the C/C++ files and to build an abstract syntax tree (AST). The type-checker of the CBMC's front-end annotates this AST with types and generates a symbol table. CBMC's IRep class then converts the annotated AST into an internal, language-independent format used by the remaining phase of the CBMC front-end.

CBMC derives the VCs using two recursive functions that compute the *assumptions* or *constraints* (i.e., variable assignments) and *properties* (i.e., safety conditions and user-defined assertions). [2] CBMC's VC generator (VCG) automatically generates safety conditions that check for arithmetic overflow and underflow, array bounds violations, and null-pointer dereferences. Both functions accumulate the control flow predicates to each program point and use that to guard both the constraints and the properties, so that they properly reflect the program's semantics. Figure 2.6 shows the CBMC architecture.

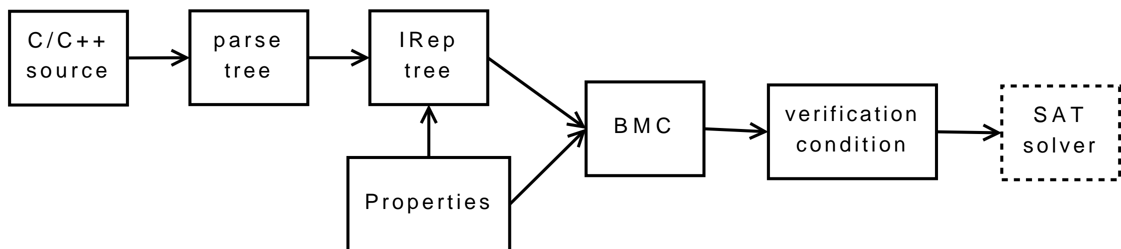

Figure 2.6: The CBMC Architecture.

Although CBMC implements several state-of-the-art techniques for propositional BMC, it still has the following limitations [11, 71]: *(i)* large data-paths involving complex expressions lead to large propositional formulae due to the number of variables and the width of data types, *(ii)* high-level information is lost when the VCs are converted into propositional logic, and *(iii)* the size of the encoding increases with the size of the arrays used in the program.

As an example of the verification process supported by CBMC, Figure 2.7 shows a syntactically valid C program that writes accidentally to an address outside the allocated

[2] Section 2.2.2 shows in a nutshell how to construct logical formulae (or VCs) from a program and desired correctness properties (for further references, we refer the reader to [8, 26, 42, 106]).

memory region of the array a (line 6). In order to reason about this C program, seven VCs are generated as follows: the first six VCs check the lower and upper bound of array a in lines 4, 6 and 7 respectively and the last VC checks the assertion stated by the user in line 7. However, before actually checking the properties, the front-end of CBMC performs a set of transformations and converts the program into single static assignment form, which requires a pointer analysis.[3]

As a result, the original C program in Figure 2.7(a), when converted into SSA form, consists only of *if*-instructions, assignments and assertions as shown in Figure 2.7(b). Note that the store operator *WITH* shown in Figure 2.7(b) takes array a, index i, and value v and produces an array a' that is exactly the same as array a, except for the content of $a[i]$ being replaced by v (e.g., $a' = a\ WITH\ [i := v]$, where a' is the modified array after the store operation) [43]. Figure 2.7(c) shows the counterexample produced by the CBMC model checker for the C program in Figure 2.7(a). In this counterexample, the content for array a is $\{0, 0\}$, which violates the invariant $a[i+1] = 1$ given that $i = 0$.

F-SOFT [96] is another tool that implements BMC for C programs, which is able to unwind the entire program. It uses the CIL front-end [140] to parse and simplify the C code (e.g., re-write *switch* and *while* in terms of *if* and *goto* statements). F-SOFT also performs a set of static analysis techniques (e.g., program slicing and range analysis [184]) to reduce the size of the unwound (bit-vector) formula. It also features a SAT solver that is highly customized to solve the verification problems arising from BMC. The main architectural difference between F-SOFT and CBMC is that the latter is able to perform program slicing with respect to the property being checked, but it lacks the implementation of range analysis to statically determine possible ranges for values of program variables. Moreover, F-SOFT adopts a block-based approach instead of a statement-based approach (as in CBMC) to model the software and it is thus able to get up to 25% of performance improvement if compared to CBMC [96]. The idea of the block-based approach is to group related statements in a block so that irrelevant blocks (i.e., those that can not affect the program's ability to reach the error block) are simply removed by backward slicing.[4] However, the developers of F-SOFT have shown benchmarks related to system-level UNIX applications (e.g., pppd), but no realistic examples of embedded software verification have been reported in the literature (and there is also no quantitative comparison of F-SOFT against CBMC) [96].

2.2.5 Comparison to Other Verification Approaches

Modern software verification tools usually make use of logic to describe states and transformations between system states. In [76], Godefroid et al. propose an SMT-based ap-

[3] Here, we omit the full details of the process to translate from ANSI-C to SSA form. For further information about this process and the CBMC model checker, we refer the reader to [46].

[4] The slice is performed by working backwards from the property of interest to the initial program location, i.e., by finding all blocks that can affect the property of interest and discarding the others [118].

```c
int main() {
  int a[2], i=0, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;          //array bounds violation
  assert(a[i+1]==1);  //violated assertion
}
```

(a)

```
i1 == 0
g1 == (x1 == 0)
a1 == (a0 WITH [0:=0])
a2 == a0
a3 == (a2 WITH [2:=1])
a4 == (g1 ? a1 : a3)
t1 == (a4[1] == 1)
```

(b)

```
Counterexample:

State 2 file <built-in> line 19 thread 0
----------------------------------------------------
  __CPROVER_alloc=(assignment removed)

State 3 file <built-in> line 20 thread 0
----------------------------------------------------
  __CPROVER_alloc_size=(assignment removed)

State 4 file <built-in> line 29 thread 0
----------------------------------------------------
  __CPROVER_rounding_mode=0 (00000000000000000000000000000000)

State 6 file example_t.c line 2 function main thread 0
----------------------------------------------------
  example_t::main::1::i=0 (00000000000000000000000000000000)

State 9 file example_t.c line 6 function main thread 0
----------------------------------------------------
  example_t::main::1::a={ 0, 0 }

Violated property:
  file example_t.c line 7 function main
  assertion
  a[i + 1] == 1
```

(c)

Figure 2.7: (a) A C program with violated property. (b) The C program of (a) in SSA form. (c) Counterexample of C program in (a)

proach called "dynamic" symbolic execution, which extends "static" symbolic execution by exploiting concrete execution paths to obtain symbolic constraints. The basic idea of dynamic symbolic execution is to explore different execution paths by selecting and negating a given branch condition from the symbolic traces. After performing this modification, the resulting path condition is encoded using the background theories that are typically supported by SMT solvers and checked for satisfiability. If the modified path condition is satisfiable, then the SMT solver provides a satisfying assignment that can be used to guide the execution through new paths. In another related work, Sen proposes an approach to execute a program concretely and symbolically by combining random testing and symbolic execution [160]. Both approaches, however, might fail to compute concrete values that satisfy a given (large) path constraint (which might be involve complex expressions) due to the solver performance.

Recently, a number of static checkers have been developed that trade off scalability and precision. PREfix is a static program analysis tool that integrates an SMT solver to perform bit-precise static analysis [34]. PREfix has been developed and used at Microsoft to analyze large C/C++ programs. Although PREfix could detect several software bugs related to arithmetic overflow in the Microsoft products, it may also detect false positive arithmetic overflow bugs as pointed out in [26]. Calysto [13] and Saturn [180] are also representative examples of static checker that employ SAT/SMT solvers as back-ends to solve the verification conditions. These tools, however, do not support fixed-point operations and are not able to detect buffer overflow bugs (which is the number one issue as reported in [3]), because they unsoundly approximates loops by unwinding them only once or twice. As a consequence of this decision, soundness is evidently relinquished for performance gains.

In extended static checking, a verification condition generator (VCG) is used to convert code annotated with "contracts" into logical formulas. The contracts consist of a pre-condition assumption inserted at some location in the program that specifies how a procedure may be called, a post-condition assertion that specifies the resulting state of a procedure call (i.e., specifies a property that has to hold at the respective location), and a loop invariant that specifies properties of intermediary system state. The Spec# programming system is a good example of a tool that integrates contracts for extended type safety [17]. Spec# uses the low-level procedural language of Boogie [18] to generate the VCs and the SMT solver Z3 [57] to check the validity of these VCs. The development of Boogie and Spec# were essentially inspired by the experiences obtained with the extended static checker ESC/Java [62]. However, in contrast to Spec#, ECS/Java employs the Simplify theorem prover [62] to verify user-supplied invariants and thus important constructs of the programming language (e.g., bitwise operation) are often encoded imprecisely using axioms and uninterpreted functions.

Explicit-state model checking is an automated technique that, given a model and a property, systematically checks whether this property holds for a given state in that

model [14]. It manipulates each state individually as opposed to symbolic model checking, which implicitly manipulates large sets of states (by applying data structures such as BDDs or SAT/SMT procedures). State space reduction techniques such as partial-order reduction thus takes advantage of the explicit-state model checking technique because it is much easier to capture and exploit transitions that are independent with respect to individual states than for a set of states [103]. In this scenario, explicit state model checkers for concurrent programs have been widely used to verify large designs that arise from the industry.

One of the most robust explicit state model checkers is Spin [90, 91], which is able to verify software models using a high level specification language called Promela (Spin also supports the use of embedded C code as part of the Promela code to verify directly low-level software). Spin implements a number of advanced optimization techniques to tackle the state explosion by using a compact representation of the search space and to reduce the number of interleavings by means of partial-order reduction. The main state compression techniques implemented in Spin include *collapse compression* (to avoid replicating a complete description of all local components of the system state) and *bitstate hashing* (to store a single bit at the slot indexed by the hash number of the state to memorize whether the corresponding state has been explored). Additionally, Spin exploits the use of multi-core computers to leverage parallelism in very large verification models.

Java Pathfinder (JPF) is another widely used explicit state model checker, which targets efficient Java bytecode verification [176]; the latest version of JPF also support symbolic model checking of Java bytecode [185]. JPF implements a set of techniques such as *backtracking* (to find different possible execution paths that have not been explored), *state matching* (to check whether every new state has already been explored), *partial order reduction* (to reduce the number of thread interleavings) and *configurable search strategies* (to use heuristics to order and filter the set of states according to the property being checked). JPF is able to check properties related to data race condition, deadlocks, heap bounds, unhandled exceptions (e.g. nil-pointer exceptions) and user-specified assertions arising from (concurrent) Java programs. For a recent survey on software model checking we refer the reader to [100].

2.3 Verification of Multi-threaded Systems

Multi-threaded software is typically difficult to validate with testing methods, mainly due to two reasons: the non-deterministic executions of the program and the potentially large state space. On the one hand, as mentioned in Chapter 1, traditional validation of multi-threaded software aims to test all possible interleaving sequences with the cost of overloading the system without ensuring complete coverage. On the other hand,

model checking multi-threaded software can guarantee complete coverage with the cost of generating an extremely large state space.

In Section 2.2, we introduced the notion of transition systems and we have shown the model checking problem associated with BMC of single-threaded programs. In BMC of multi-threaded programs, we still have the same notion of states (i.e., the assignment of values to variables) as described in the sequential case, but here we must now consider the interleavings of transitions of different threads. In particular, a multi-threaded program contains a number of threads that execute in parallel and the execution of a thread is scheduled in a non-deterministic way by a global system scheduler, as will be explained in the next section. For now, we informally assume that the operational behaviour of the threads that run in parallel are given by transitions systems $M_1, \ldots, M_n$ (recall Definition 2.14). We can then define a transition system $M_t = \bigcup_{j=0}^{n} M_j$ that specifies the behaviour of the parallel composition of transition systems M_1 through M_n.

This section describes mechanisms to model multi-threaded systems by means of transition systems composed from different individual threads; further information can be found in textbooks [14, 40]. This then allows us to encode explicitly the interleaving model into the BMC framework to model check multi-threaded programs.

2.3.1 Concurrency and Interleaving

There are two modes of concurrent execution; *asynchronous* and *synchronous*. In the asynchronous mode, which we consider, only one thread can make progress at a time, whereas in the synchronous mode all threads can run at the same time. Threads in asynchronous mode can communicate via message passing or shared variable. In the message passing model, threads can send/receive messages (comprising zero or more bytes, data structures, or even segments of code) to/from other threads. In the shared variable model, a region of memory may be simultaneously accessed by multiple threads in order to provide communication among them.

Thread synchronization or serialization (e.g., via mutual exclusion or condition variable) ensures that multiple threads do not access specific regions of memory at the same time. This means that if one thread started to access a region of memory, any other thread trying to access this region must wait until the first thread finishes. This work considers multi-threaded programs with asynchronous mode and assumes that the threads in the program only communicate through shared (global) variables and synchronize to avoid the simultaneous access to shared variables. Note that this assumption also applies to the verification of software in multi-core systems since asynchronous operation is a standard solution to avoid contention for memory in multi-core processors [60].

A widely adopted paradigm for multi-threaded programs is that of *interleaving*. [5] In

[5] The definition of interleaving is based on the notion of the asynchronous mode, i.e., only one thread

this paradigm, an *interleaving sequence* represents a possible execution of the program where all of the concurrent events are arranged in a linear order. Thus, the notion of concurrency is represented by that of interleaving, that is, the non-deterministic choice between activities of the simultaneously acting threads. This perspective is based on the fact that only one core is available on which the actions of the threads are interleaved. From the modelling point of view, this concept also applies if the threads run on different cores. In both cases (single-core or multi-core), there are many interleaving sequences with different orderings between concurrent events.

The interleaving representation of concurrency depends on a scheduler, which interleaves the steps of concurrently executing threads according to a given strategy. This type of representation completely abstracts from the speed of the participating threads and thus models any possible realization by a single-core machine or by several cores with arbitrary speeds. From the verification point of view, in order to fully verify a concurrent program against a given specification, all possible interleaving sequences must be considered. This can result in an extremely large state space that must be explored by a model checker, which in turn is the main source of state explosion problem.

As a running example, consider the control-flow graph (CFG) of two threads, say T_A and T_B as shown in Figure 2.8, where variables a and b are declared as global. For each thread T_i, its control-flow graph is a directed graph $T_i = \langle N_i, E_i, n_{i0} \rangle$, where N_i is the set of nodes that represent program statements, E_i is the set of edges that represent transitions (i.e., saying how each thread T_i can move from node to node) and n_{i0} is the initial node. In our example, thread $T_A = \langle N_A, E_A, n_{A_0} \rangle$ where the nodes $N_A = \{T_{A0}, T_{A1}, T_{A2}, T_{A3}\}$, the edges $E_A = (T_{A0} \rightarrow T_{A1}, T_{A1} \rightarrow T_{A2}, T_{A2} \rightarrow T_{A3})$, and the initial node $n_{A_0} = T_{A0}$, while thread $T_B = \langle N_B, E_B, n_{B_0} \rangle$ where the nodes $N_B = \{T_{B0}, T_{B1}, T_{B2}, T_{B3}\}$, the edges $E_B = (T_{B0} \rightarrow T_{B1}, T_{B1} \rightarrow T_{B2}, T_{B2} \rightarrow T_{B3})$, and the initial node $n_{B_0} = T_{B0}$.

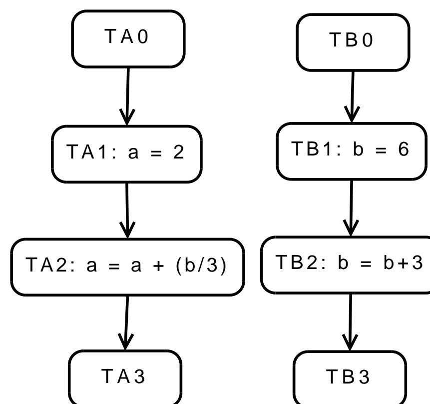

FIGURE 2.8: The CFG representation of threads T_A and T_B and we assume that initially the global variables a and b are set to zero, i.e., $a = 0$ and $b = 0$.

We say that a program statement is *visible* if it accesses a global variable, and it is *invisible* otherwise. In our example, we consider that all program statements (i.e., $a = 2$,

is executed at a given time.

$a = a + (b/3)$, $b = 6$ and $b = b + 3$) are visible.

An interleaving represents a possible execution of the program where all of the concurrent events are arranged in a linear order. Any change of the active thread in an interleaving is a *context switch*. A program statement is considered to be *atomic* if no context switch can happen during its execution. Statements that involve at most one global variable are not affected by context switches. In our example, program statements $a = 2$, $b = 6$ and $b = b + 3$ are atomic while the program statement $a = a + (b/3)$ is not atomic, because it is affected by context switches.

The CFG that represents all possible interleaving sequences of threads T_A and T_B is shown in Figure 2.9. The number of possible interleaving sequences I for a given number of threads N consisting of s program statements in a program without loops can be computed as follows [176].

$$I = \frac{\left(\sum_{i=1}^{N} s_i\right)!}{\prod_{i=1}^{N}(s_i!)} \tag{2.15}$$

In our running example, we have $N = 2$, $s_A = 2$ and $s_B = 2$ and the number of possible interleaving sequences is thus:

$$I = \frac{(2+2)!}{(2! \cdot 2!)} = \frac{24}{6} = 6 \tag{2.16}$$

The transition system that represents the parallel execution of threads T_A and T_B is shown in Figure 2.10. As we can see in Figure 2.10, the choice of two (i.e., those that have as final state $\{a = 4, b = 9\}$) and three (i.e., those that have as final state $\{a = 5, b = 9\}$) interleaving sequences of the threads in Figure 2.8 do not affect the final state (i.e., they result in the same state when executed in different orders) and so they generate equivalent interleaving sequences. Unfortunately, this observation is not true for the example in Figure 2.8, because we have to consider context switches inside the individual visible statements that involve more than one access to a global variable (since threads T_A and T_B share the same global variable b). For example, the program statement $a = a + (b/3)$ in Figure 2.8 is thus broken into three different statements (see nodes $T_{A'_2}$, $T_{A'_3}$, and $T_{A'_4}$ in Figure 2.11) so that a context switch may now occur between these statements. In chapter 4, we show how to break the visible program statements and check for atomicity violations. In this new scenario, the number of possible interleaving sequences increases from six (without considering context switches inside the individual visible statements) to fifteen as follows:

$$I = \frac{(4+2)!}{(4! \cdot 2!)} = \frac{720}{48} = 15 \tag{2.17}$$

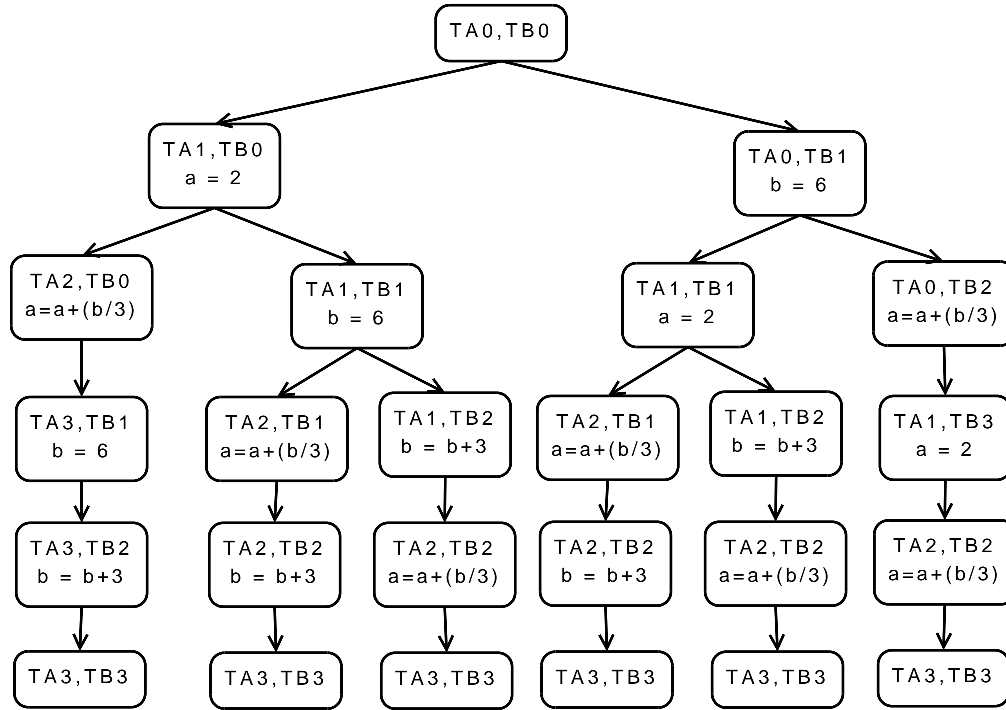

FIGURE 2.9: The CFG that represents all possible interleaving sequences of threads T_A and T_B.

However, in order to remove redundant interleaving sequences, partial order reductions are usually applied to reduce significantly the size of the traversed model (i.e., the number of possible interleavings to be checked).

2.3.2 Partial Order Reduction Technique

The name Partial Order Reduction (POR) comes from partial order model of program execution [75]. According to the model, concurrently executed events are not ordered and each partially ordered execution can correspond to multiple interleaving sequences. In [146], the name model checking using representatives is used to better describe the name partial order reduction since the verification is carried out using representatives from equivalence classes of the behaviours. POR techniques [14, 40, 49, 103, 131] aim to prune the number of states that have to be searched by model checking algorithms. This is done by removing interleaving sequences that lead to the same system state, i.e., it avoids exploring different equivalent interleavings of the concurrent events.

As an example of the number of states to be searched, consider the parallel composition of a number of threads T_1 through T_n. The size of the state space to be explored, which consists of the parallel composition of transition systems M_1 through M_n (i.e., $M_t = \bigcup_{j=0}^{n} M_j$), is exponential in the number n of threads and program statements. To model check a simple LTL property of this system requires an inspection of all states in the underlying transition system M_t. However, instead of constructing a full state

FIGURE 2.10: The transition system that represents the parallel execution of threads T_A and T_B.

FIGURE 2.11: Model context switches inside individual visible statements

graph, which may be too large to fit in memory, POR techniques aim to build a reduced state graph using only representatives from the equivalence classes of behaviours.

Naturally, the POR techniques are best suited if applied to concurrent asynchronous system, because there we can exploit the commutativity of concurrently executed independent events, i.e., events that result in the same state when executed in different

orders. In this scenario, POR is done in a way that if the property ϕ holds on the reduced model, say $M' = (S', R', S0)$, it also holds on the original model $M = (S, R, S0)$ (recall Definition 2.14). This reduction is then based on the notion of independence relation between transitions ($I \subseteq R \times R$), which is defined as follows.

Definition 2.19. *$I \subseteq R \times R$ is an independence relation if and only if for each α and β, where $(\alpha, \beta) \in I$, the following two conditions hold for all $s \in S$:*

1. *Transitions α and β may execute in either order from state s, i.e., if α is enabled in s and $s \xrightarrow{\alpha} s'$, then β is enabled in s if and only if β is enabled in s';*

2. *Executing either of the two transition α and β starting from state s leads to the same state s', i.e., if α and β are enabled in s, there is a unique state s' such that $s \xrightarrow{\alpha,\beta} s'$ and $s \xrightarrow{\beta,\alpha} s'$.*

The intuitive interpretation of these two conditions is that (1) independent transitions can neither disable nor enable each other (enabledness), and (2) executing them in either order results in the same state (commutativity). The dependency relation D is simply defined as the complement of I, i.e., if two transitions α and β are not independent, then they are dependent. The partial order reduction thus exploits the dependency relation that exists between the transitions of the threads. From a pragmatic point of view, two transitions α (related to thread T_1) and β (related to thread T_2) are called to be independent of each other if and only if the execution of α and β in either order results in the same global state. If interleaving sequences that differ only by such independently executed events are indistinguishable by a specification, they are called to be equivalent. It is thus sufficient to select only one interleaving sequence from such equivalence class as representative to be checked against the specification by a model checking algorithm.

Classic POR algorithms explore at each state s an adequate subset $ample(s)$ of the transitions enabled (the set of transitions enabled in s is denoted by $enabled(s)$). This exploration has to respect a set of conditions based on Definition 2.19:

- **Condition C0:** $ample(s) = \emptyset$ *iff* $enable(s) = \emptyset$.

- **Condition C1:** Along every path of the (full) state graph starting in s, a transition that is dependent on a transition α in $ample(s)$ must be preceded by α, i.e, transition α has to occur first.

- **Condition C2:** if $ample(s) \neq enabled(s)$, then each transition α in $ample(s)$ must be invisible w.r.t. property ϕ.

- **Condition C3:** If for each state $s \in S$ of a cycle in reduced model M', a transition α is enabled, then α must be in $ample(s)$.

Conditions C0 to C3 are sufficient to guarantee that the resulting (reduced) model M' preserves properties specified in LTL (described in Section 2.2) [14, 40].

2.4 Summary

This chapter described the main concepts needed to understand this thesis. In Section 2.1, *Logical Foundations*, we introduced PL syntax and semantics along with some examples. We defined a mechanism to check whether a given PL formula is *true* or *false* by means of *interpretations*. In particular, we showed that given a PL formula and an interpretation, the truth value of a formula can be computed by a truth table (most commonly used for evaluating PL formulae) or by induction (which is most suitable for evaluating first-order logic formulae). We also described the problem of deciding the satisfiability of PL formulae and how SAT solvers deal with this problem. In Subsection 2.1.3, we described the SMT problem, which aims to decide the satisfiability of first-order logic formulae using a combination of different background theories. We also presented the main background theories implemented in modern SMT solvers and their advantages over SAT solvers when reasoning about verification problems arising from real-world applications. In Subsection 2.1.4, we also described together with some illustrative examples the specification logic LTL that is commonly used to specify properties in the BMC framework.

In Section 2.2, *Bounded Model Checking of Software*, we presented the BMC technique that consists of unwinding the design and the correctness property k times, and generating a propositional formula that is satisfiable if and only if a counterexample exists. We also discussed that the BMC technique can be used to find violations of the temporal property up to the bound k, but not to prove properties. In Subsection 2.2.3, we described two methods to *prove* properties in the BMC framework, which are Craig interpolants and k-induction. Craig interpolation in model checking exploits the SAT/SMT solvers' ability to produce proof of unsatisfiability. This proof does not ensure whether a given property holds in the model, but it contains information about the reachable states of the model. Therefore, the use of interpolants allows us to define a complete method for finite-state reachability analysis based entirely on SAT and SMT solvers. The k-induction method is a stronger version of the standard invariant approach to verify safety properties. We present it as temporal induction (i.e., the induction is carried out over the time steps of the finite state machines) and we also showed how to devise an algorithm from the k-induction method to prove properties in the BMC framework. We also overviewed typical architectures of the BMC technique such as those implemented in the CBMC and F-SOFT model checkers, which are able to model check ANSI-C programs. We conclude this section by comparing the BMC technique to other modern software verification techniques that make use of logic to describe states and transformations between system states.

Finally, in Section 2.3, we provided mechanisms to model multi-threaded systems by means of transition systems, which allow us to encode multi-threaded systems into the BMC framework. We also presented the concept of asynchronous and synchronous modes where the former only allows one thread to make progress at a time, and the latter allows all threads to run at the same time. In this sense, we further presented the message passing and shared variable models. In the message passing model, threads can send/receive messages to/from other threads; while in the shared variable model, a region of memory may be simultaneously accessed by multiple threads in order to provide communication among them. As in this work we focus on asynchronous systems, we then described the interleaving paradigm to model multi-threaded programs, which represents a possible execution of the program where all of the concurrent events are arranged in a linear order. We thus concluded this section by showing the effectiveness of partial order reduction techniques to prune the number of states that have to be searched by model checking algorithms.

Chapter 3

SMT-based Bounded Model Checking for Embedded ANSI-C Software

Propositional bounded model checking has been applied successfully to verify embedded software but remains limited by increasing propositional formula sizes and the loss of high-level information during the translation preventing potential optimizations to reduce the state space to be explored. These limitations can be overcome by encoding word-level information in theories richer than propositional logic and using SMT solvers for the generated verification conditions. Here, in order to achieve the first objective stated in Section 1.2, we have modified and extended the encodings from previous SMT-based bounded model checkers to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers. Additionally, to achieve that objective, we have integrated the Boolector [31], CVC3 [20], and Z3 [57] solvers with the CProver framework and evaluated them using both standard software model checking benchmarks and typical embedded software applications from telecommunications, control systems, and medical devices. The experiments show that our ESBMC model checker can analyze larger problems than existing tools and substantially reduce the verification time.

3.1 Introduction

Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) has been introduced as a complementary technique to Binary Decision Diagrams (BDDs) for alleviating the state explosion problem [24]. The basic idea of BMC is to check the negation of a given property at a given depth: given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a verification condition

(VC) ψ such that ψ is satisfiable if and only if ϕ has a counterexample of depth k or less. Standard SAT checkers can be used to check whether ψ is satisfiable. Note that in BMC of software, the bound k limits the number of loop iterations and recursive calls in the program.

In order to cope with increasing software complexity, SMT (Satisfiability Modulo Theories) solvers can be used as back-ends for solving the generated VCs [10, 11, 71, 105]. Here, predicates from various decidable theories are not encoded using propositional variables as in SAT, but remain in the problem formulation. These theories are handled by dedicated decision procedures. Thus, in SMT-based BMC, ψ is a quantifier-free formula in a decidable subset of first-order logic which is then checked for satisfiability by an SMT solver.

In order to reason about embedded software accurately, an SMT-based BMC must consider a number of issues that are not easily mapped into the theories supported by SMT solvers. In previous work on SMT-based BMC for software [10, 11, 71] only the theories of uninterpreted functions, arrays and linear arithmetic were considered, but no encoding was provided for ANSI-C [95] constructs such as bit-level operations, fixed-point arithmetic, pointers (i.e., pointer arithmetic and comparisons) and unions. This limits its usefulness for analyzing and verifying embedded software written in ANSI-C. In addition, the SMT-based BMC approaches proposed by Armando et al. [10, 11] and by Kroening [105] do not support the checking of arithmetic overflow and do not make use of high-level information to simplify the unrolled formula. We address these limitations by exploiting the different background theories of SMT solvers to build an SMT-based BMC tool that precisely translates program expressions into quantifier-free formulae and applies a set of optimization techniques to prevent overburdening the solver. This way we achieve significant performance improvements over SAT-based BMC and the previous work on SMT-based BMC [10, 11, 71, 105].

We describe the details of an accurate translation from single-threaded ANSI-C programs into quantifier-free formulae using the logics QF_AUFBV and QF_AUFLIRA from the SMT-LIB [164].

Definition 3.1. The QF_AUFBV logic represents quantifier-free formulae that are built over bit-vectors and arrays with free sort and function symbols, but with the restriction that all array terms have the following structure (array (bit-vector i[w_1]) (bit-vector v[w_2])), where i is the index with bit-width w_1 and v is the value with bit-width w_2.

Definition 3.2. The QF_AUFLIRA logic represents quantifier-free formulae that are built over reals, integers and arrays with free sort and function symbols, but with the restriction that all array terms are of the sort (array int real) or (array int (array int real)), where all argument terms of sort int and real are linear, i.e., there is no occurrences of the function symbols $*$, $/$, div, rem, and abs.

We further demonstrate that our encoding and optimizations improve the performance of software model checking for a wide range of software systems, with a particular emphasis on embedded software. Additionally, we show that our encoding allows us to reason about arithmetic overflow and to verify programs that make use of bit-level, pointers, unions and fixed-point arithmetic. We also use three different SMT solvers (Boolector [31], CVC3 [20], and Z3 [57]) in order to check the effectiveness of our encoding techniques. We considered these solvers because they were the most efficient ones for the categories of QF_AUFBV and QF_AUFLIRA in the last SMT competitions [168]. To the best of our knowledge, this is the first work that reasons accurately about ANSI-C constructs commonly found in embedded software and extensively applies SMT solvers to check the VCs emerging from the BMC of industrial embedded software applications. We implemented our ideas in the ESBMC[1] (Efficient SMT-Based Bounded Model Checker) tool that builds on the front-end of the C Bounded Model Checker (CBMC) [42, 107]. ESBMC supports different theories and SMT solvers in order to exploit high-level information to simplify and to reduce the formula size. Experimental results show that our approach scales significantly better than both the SAT-based and SMT-based CBMC model checker [42, 107, 105] and SMT-CBMC [11], a bounded model checker for C programs that is based on the SMT solvers CVC3 and Yices.

The remainder of the chapter is organized as follows. In Section 3.2 we describe the SMT-based BMC Formulation. In Section 3.3 we provide a running example to illustrate our encoding while in Section 3.4 we present the details of an accurate translation from ANSI-C programs into quantifier-free formulae using the SMT logics. In Section 3.5 we present the results of our experiments using several software model checking benchmarks and embedded systems applications while in Section 3.6 we describe the results of applying ESBMC to the verification of a commercial embedded software used in the telecommunications domain. In Section 3.7 we discuss the related work and we conclude and describe future work in Section 3.8.

3.2 SMT-based BMC Formulation

In BMC, the program to be analyzed is modelled as a state transition system, which is extracted from the control-flow graph (CFG) [134]. This graph is built as part of a translation process from program text to single static assignment (SSA) form. A node in the CFG represents either a (non-) deterministic assignment or a conditional statement, while an edge in the CFG represents a possible change in the program's control location.

Let M be an abstract machine that represents a state transition system according to Definition 2.14. A state $s \in S$ consists of the value of the program counter pc and the values of all program variables. An initial state s_0 assigns the initial program location of

[1] Available at http://users.ecs.soton.ac.uk/lcc08r/esbmc/

the CFG to *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a VC ψ such that ψ is satisfiable if and only if ϕ has a counter-example of length k or less. The VC ψ is a quantifier-free formula in a decidable subset of first-order logic, which is then checked for satisfiability by an SMT solver. In this chapter, we are interested in checking safety properties of single-threaded programs. The associated model checking problem is formulated by constructing the following logical formula:

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{3.1}$$

Here, ϕ is a safety property, I the set of initial states of M and $\gamma(s_j, s_{j+1})$ the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the executions of M of length i and ψ_k can be satisfied if and only if for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If ψ_k is satisfiable, then ϕ is violated and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, $s_k \in S$, and $\gamma(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k is unsatisfiable, we can conclude that no error state is reachable in k steps or less. Note that formula (3.1) differs slightly from (2.11) (presented in Section 2.2) because it represents a violation of length k or less to the considered safety property while (2.11) represents a violation of exactly length k. This means that if the system deadlocks in $l \leq k$ steps and the error is at step $j \leq l$, then the formula (2.11) turns out to be unsatisfiable and therefore it will not detect the error.

It is important to note that this approach can be used only to find violations of the property up to the bound k. In order to *prove* properties we need to compute the *completeness threshold* (CT), which can be smaller than or equal to the maximum number of loop-iterations occurring in the program [24, 45, 72]. However, computing CT to stop the BMC procedure and to conclude that no counter-example can be found is as hard as model checking. Moreover, complex programs involve large data-paths and complex expressions. Consequently, even if we knew CT, the resulting formulae would quickly become too hard to solve and require too much memory to build. In practice we can thus only ensure that the property holds in M up to a given bound k. In our work, we focus on embedded software because it has characteristics that make it attractive for BMC, e.g., dynamic memory allocations and recursion are highly discouraged, and that make the limitations of *bounded* model checking less stringent.

3.3 Illustrative Example

We use the code shown in Figure 3.1 as a running example to illustrate the process of transforming a given ANSI-C program into SSA form and then into the quantifier-free formulae C and P shown in formulas (3.2) and (3.3). This code implements a simplified version of the *character stuffing* technique, which avoids resynchronization after an error by starting each frame with the ASCII character sequence DLE STX and ending it with the sequence DLE ETX [169]. Note that this syntactically valid ANSI-C program contains two subtle errors. One error is that it writes in line 28 to an address outside the allocated memory region of the array *out*. The second error occurs when the ASCII character "NULL" is transmitted, i.e., the condition of the *while* loop (line 11) does not hold; as a result the `assert` macro in line 29 fails. To detect this error, we use a non-deterministic input, i.e., we set the third position of array *in* (line 6) to *nd_uchar*(), which can return any value in the range from zero to 255.

```
1  #define DLE 16
2  #define STX 2
3  #define ETX 3
4  uchar nd_uchar();
5  int main (void) {
6    uchar in[6] = {DLE, STX, nd_uchar(),
7                   DLE, ETX, '\0'};
8    uchar out[6];
9    int i = 0;
10   int j = 0;
11   while (in[i] != '\0') {
12     switch (in[i]) {
13       case (DLE):
14         if (in[i+1]==STX || in[i+1]==ETX) {
15           out[j] = in[i];
16         } else {
17           out[j] = in[i];
18           out[++j] = DLE;
19         };
20         break;
21       default:
22         out[j] = in[i];
23         break;
24     }
25     i++;
26     j++;
27   }
28   out[j] = '\0';
29   assert(out[4]==ETX || out[5]==ETX);
30   return 0;
31 }
```

FIGURE 3.1: ANSI-C program with two violated properties.

In reasoning about this C program, ESBMC checks 25 properties (or *claims*) related to array bounds and overflow, and the user-specified assertion in line 29. However, before

actually checking the claims, the front-end unrolls the program using the simplification described in Section 5.3 and converts it into SSA form, which only consists of conditional and unconditional assignments as well as assertions, as shown in Figure 3.2. For each assignment (e.g., $i = 0$), the left-hand side variable is replaced by a new variable (e.g., i_1). In addition, in Figure 3.2 the variable declarations as well as the return-statement are removed. The SSA notation uses WITH as symbolic representation of the array *store* operator described in Section 2.1.3, i.e., a WITH $[i := v]$ is equivalent to *store(a, i, v)*. After unrolling, ESBMC initially generates 63 VCs, but after the simplifications described in detail in Section 5.3, only 9 remain. The first eight of these VCs check the bounds of the array *out* in lines 15, 18 and 28 and the last VC checks the user-specified assertion in line 29; note that the VCs to check the bounds of the array *out* are not simplified away due to the non-determinism in one of the elements of the array *in*, which does not allow checking statically whether the guard of the *if* statement in line 14 is *true* or *false*. For comparison, in this particular example, CBMC v3.8 generates 136 VCs out of which 48 remain after simplification. The limited static analysis capability of CBMC thus leads to a substantially higher overhead in the solver.

```
1  in1 == {16, 2, nd_uchar1, 16, 3, 0}
2  i1 == 0
3  j1 == 0
4  out1 == (out0 WITH [0:=16])
5  i2 == 1
6  j2 == 1
7  out2 == (out1 WITH [1:=2])
8  i3 == 2
9  j3 == 2
10 g1 == (nd_uchar1 != 0)
11 g2 == !(nd_uchar1 == 16)
12 out3 == (out2 WITH [2:=nd_uchar1])
13 j4 == 3
14 out4 == (out3 WITH [3:=16])
15 out5 == out2
16 j5 == j3
17 out6 == (out5 WITH [j5:=nd_uchar1])
18 out7 == (!g2 ? out4 : out6)
19 j6 == (!g2 ? j4 : j5)
20 i4 == 3
21 j7 == 1 + j6
22 out8 == (out7 WITH [j7:=16])
23 i5 == 4
24 j8 == 1 + j7
25 out9 == (out8 WITH [j8:=3])
26 i6 == 5
27 j9 == 1 + j8
28 out10 == (!g1 ? out2 : out9)
29 i7 == (!g1 ? i3 : i6)
30 j10 == (!g1 ? j3 : j9)
31 out11 == (out10 WITH [j10:=0])
```

FIGURE 3.2: The program of Figure 3.1 in SSA form.

$$C := \left[\begin{array}{l} in_1 = store(store(store(store(store(store(in_0, \\ 0, 16), \\ 1, 2), \\ 2, nd_uchar_1), \\ 3, 16), \\ 4, 3), \\ 5, 0) \\ \wedge\ i_1 = 0 \wedge j_1 = 0 \ \wedge\ out_1 = store\,(out_0, 0, 16) \\ \wedge\ i_2 = 1 \wedge j_2 = 1 \ \wedge\ out_2 = store\,(out_1, 1, 2) \\ \wedge\ g_1 = nd_uchar_1 \neq 0 \\ \wedge\ g_2 = \neg\,(nd_uchar_1 = 16) \\ \wedge\ out_3 = store\,(out_2, 2, nd_uchar_1) \\ \wedge\ j_4 = 3 \\ \wedge\ \ldots \\ \wedge\ j_{10} = ite\,(\neg g_1, j_3, j_9) \\ \wedge\ out_{11} = store\,(out_{10}, j_{10}, 0) \end{array} \right] \tag{3.2}$$

$$P := \left[\begin{array}{l} j_5 \geq 0 \wedge j_5 < 6 \ \wedge\ j_7 \geq 0 \wedge j_7 < 6 \\ \wedge\ j_8 \geq 0 \wedge j_8 < 6 \ \wedge\ j_{10} \geq 0 \wedge j_{10} < 6 \\ \wedge\ \ ((select\,(out_{11}, 4) = 3) \vee (select\,(out_{11}, 5) = 3)) \end{array} \right] \tag{3.3}$$

After this transformation, we build the constraints and properties as shown in formulae (3.2) and (3.3) using the background theories of the SMT solvers. Furthermore, we create additional Boolean variables (called *definition literals*) for each clause of the formula P in such a way that the definition literal is true if and only if a given clause of the formula P is true. In the example we add a constraint for each clause of P as follows:

$$\begin{array}{l} l_0 \Leftrightarrow j_5 \geq 0 \\ l_1 \Leftrightarrow j_5 < 6 \\ \ldots \\ l_9 \Leftrightarrow ((select\,(out, 4) = 3) \vee (select\,(out, 5) = 3)) \end{array}$$

These definition literals are used to identify the VCs. Note that the language-specific safety properties (e.g., out-of-bounds array indexing) and the user-specified properties that hold trivially in the code are already simplified away (e.g., by keeping track of the size of the array during the symbolic execution of the code). For instance, there is no need to generate VCs that check for violations of the lower and upper bound of array in, since i only takes the values from 0 to 4 when it is used in indexing the array, and the validity of the bounds check can be evaluated statically. After mapping each VC to

a definition literal, we then rewrite (3.3) as:

$$\neg P := \neg l_0 \vee \neg l_1 \vee \ldots \vee \neg l_9 \tag{3.4}$$

Finally, the formula $C \wedge \neg P$ is passed to an SMT solver to check satisfiability. Our approach is thus slightly different from that of Armando et al. [11], who transform the ANSI-C code into conditional normal form as an intermediary step to encode C and P while we first apply a number of simplifications (as described in Section 5.3) during the transformation and then encode the ANSI-C code directly from the simplified SSA form. Consequently, Armando et al. [11] end up with two sets of quantifier-free formulae C and P (but possibly with a higher overhead for the solver) and check the validity $C \models_{\mathcal{T}} \bigwedge P$ using an SMT solver.

3.4 Encodings and Properties

This section describes the encodings that we use to convert the constraints and properties from the ANSI-C program into the background theories of the SMT solvers.

3.4.1 Scalar Data Types

We provide two approaches to model (unsigned and signed) integer data types, either as the integers provided by the corresponding SMT-lib theories or as bit-vectors, which are encoded using a particular bit width such as 32 bits. Table 3.1 shows a list of the ANSI-C types and their corresponding bit-vector representations, based on the storage sizes (i.e., number of bits) required by ISO ANSI-C [95]. It also gives the representation using the abstract numerical domains of the SMT-LIB.

In our SMT-based BMC framework, the encoding of the relational (e.g., $<$, $\leq$, $>$, $\geq$) and arithmetic operators (e.g., $+$, $-$, $/$, $*$, rem) then depends on the encoding of their operands as unsigned or signed bit-vectors, or integer or fixed-point numbers. The SMT-based BMC approach proposed by Armando et al. [11] does not support the encoding of fixed-point numbers and Kroening [105] does not exploit the SMT solvers to model the program variables through the corresponding numerical domain (e.g., $\mathbb{Z}$, $\mathbb{R}$). Additionally, the SAT-based BMC approach of Clarke et al. [42] (note that [42] is the original paper that describes the CBMC's implementation; a detailed technical report can be found in [107]) transform the relational and arithmetic operators into a propositional equation using a carry chain adder and the size of their encoding thus depends on the size of the bit-vector representation of the scalar data types.

For the bit-vector encodings, the front-end provides six scalar datatypes: *bool*, *signedbv*, *unsignedbv*, *fixedbv*, *floatbv*, and *pointer*. The ANSI-C datatypes *int, long int, long long*

ANSI-C Type	SMT bit-vector representation			SMT abstract numerical domain
	16-bit architecture	32-bit architecture	64-bit architecture	
bool	bool(1)	bool(1)	bool(1)	bool
char	signedbv(8)	signedbv(8)	signedbv(8)	integer
unsigned char	unsignedbv(8)	unsignedbv(8)	unsignedbv(8)	unsigned integer
short int	signedbv(16)	signedbv(16)	signedbv(16)	integer
unsigned short int	unsignedbv(16)	unsignedbv(16)	unsignedbv(16)	unsigned integer
int	signedbv(16)	signedbv(32)	signedbv(32)	integer
unsigned int	unsignedbv(16)	unsignedbv(32)	unsignedbv(32)	unsigned integer
long int	signedbv(32)	signedbv(32)	signedbv(64)	integer
unsigned long int	unsignedbv(32)	unsignedbv(32)	unsignedbv(64)	unsigned integer
long long int	signedbv(64)	signedbv(64)	signedbv(128)	integer
unsigned long long int	unsignedbv(64)	unsignedbv(64)	unsignedbv(128)	unsigned integer
pointer	pointer(32)	pointer(32)	pointer(64)	integer
double	fixedbv(64)	fixedbv(64)	fixedbv(64)	real

TABLE 3.1: Definitions of ANSI-C types and their corresponding SMT representations.

int, and *char* are considered as *signedbv* with different bit widths (depending on the machine architecture) and the unsigned versions of these datatypes are considered as *unsignedbv*. For *double and float* we currently only support fixed-point arithmetic (i.e., *fixedbv*) at this point in time, but not full floating-point arithmetic (i.e., *floatbv*); see the following section for more details.

We support all type casts, including conversion between integer and fixed-point types. In the bit-vector representation, the conversions between *signedbv*, *unsignedbv* and *fixedbv* are performed using the word-level functions $Extract\,(Trm, i, j)$, $SignExt\,(Trm, k)$ and $ZeroExt\,(Trm, k)$ described in Section 2.1.3. Similarly, upon dereferencing, the object that a pointer points to is converted using the same word-level functions. The conversions between *signedbv*, *unsignedbv* and *fixedbv* using the abstract numerical domains are straightforward; we only consider the integral part. In addition, *signedbv* and *unsignedbv* are converted to *bool* using the $\neq$-operator by comparing the variable to be converted with zero. Formally, let v be a variable of signed or unsigned type, k be a constant whose value represents zero in the type of v, and t be a Boolean variable such that $t \in \{0, 1\}$. We then convert v into t as follows:

$$t = ite\,(v \neq k, 1, 0) \tag{3.5}$$

while *bool* is converted to *signedbv* and *unsignedbv* using the *ite*-operator as follows:

$$v = ite\,(t, 1, 0) \tag{3.6}$$

As an illustrative example, consider the fragment of code (extracted from the VERISEC suite) as shown in Figure 3.3, which contains a *do-while* loop in lines 4-6 with a halt condition $str[i]$!= EOS that requires a typecast operation from *char* to *int*.

```
1 #define EOS 0
2 static int parse_expression(char *str) {
3   ...
4   do {
5   ...
6   } while (str[i] != EOS);
7 }
```

FIGURE 3.3: ANSI-C program with typecast from *char* to *int*.

In order to check the condition $str[i]$!= EOS, we must first select the i-th element of *str* and then apply a sign extension to it since EOS is of type *signedbv(32)* and *str* is of type *signedbv(8)* in a 32-bit architecture. As a result, the encoding $SignExt\,(select(str, i), 24)$ extends the i-th element of *str* to the signed equivalent bit-vector of size 32.

3.4.2 Fixed-Point Arithmetic

Embedded applications from domains such as discrete control and telecommunications often require arithmetic over non-integral numbers. However, an encoding of the full floating-point arithmetic into the BMC framework leads to large formulae; instead, we approximate it by fixed-point arithmetic, which might introduce behaviour that is not present in a real implementation. We use two different representations to encode non-integral numbers, binary (when dealing with bit-vector arithmetic) and decimal (when dealing with rational arithmetic). In this way, we can explore the different background theories of the SMT solvers and trade off speed and accuracy as further described in Section 5.4. In both encodings, we encode fixed-point numbers using the integral and fractional parts separately [109].

Binary encoding. Given a rational number that consists of an integral part I with m bits and a fractional part F with n bits, we represent it by $\langle I.F \rangle$ and interpret it as $I + F/2^n$. For instance, the number 0.75 can be represented as $\langle 0000.11 \rangle$ in base 2 while 0.125 can be represented as $\langle 0000.0010 \rangle$. We encode fixed-point arithmetic using bit-vector arithmetic as in the binary integer encoding (we concatenate the integral and fractional parts), but we assume that the operands have the same bitwidths both before and after the radix point. If this is not the case, we pad the shorter bit sequence and add zeros from the right (if there are bits missing in the fractional part) using the word-level function *ZeroExt* or from the left (if there are bits missing before the radix point) using the word-level function *SignExt*. Continuing the example, we thus get $0.75 + 0.125 = \langle 0000.1100 \rangle + \langle 0000.0010 \rangle$.

Formally, let a be a bit-vector of size t_a, with m_a and n_a (where $t_a = m_a + n_a$) the

number of bits of the integral and fractional parts, respectively, and b be a bit-vector of size t_b, with m_b and n_b defined similarly. We apply the encodings in (3.7) and (3.8) in order to get the new bit-vector $b = i@f$ that has the same bitwidth before and after the radix point of a.

$$i = \begin{cases} \mathit{Extract}\left(b, n_b + m_a - 1, n_b\right) & : \; m_a \leq m_b \\ \mathit{SignExt}\left(\mathit{Extract}\left(b, t_b - 1, nb\right), m_a - m_b\right) & : \; \mathit{otherwise} \end{cases} \tag{3.7}$$

$$f = \begin{cases} \mathit{Extract}\left(b, n_b - 1, n_b - n_a\right) & : \; n_a \leq n_b \\ \mathit{ZeroExt}\left(\mathit{Extract}\left(b, n_b - 1, 0\right), n_a - n_b\right) & : \; \mathit{otherwise} \end{cases} \tag{3.8}$$

Rational encoding. We encode fixed-point arithmetic using rational arithmetic by rounding the fixed-point numbers to rationals in base 10. We extract the integral and fractional parts and convert them to integers I and F, respectively; we then divide F by 2^n, round the result to a given number of decimal places, and convert everything to a rational number in base 10. Formally, let p be the number of decimal places and let i and f be the integral and fractional parts resp. of a given fixed-point number a. We apply the encoding in (3.9) in order to convert a to a rational number.

$$a = \begin{cases} \left(i * p + \left(\frac{f * p}{2^n} + 1\right)\right)/p & : \; f \neq 0 \\ i & : \; \mathit{otherwise} \end{cases} \tag{3.9}$$

For example, with $m = 2$, $n = 16$, and six places decimal precision, the number 3.9 (with a binary representation of 11.1110011001100110) is converted to $I = 3$, and $F = 58982/2^{16}$, and finally to 3899994/100000. As a result, the arithmetic operations are performed in the domain of $\mathbb{Q}$ instead of $\mathbb{R}$ and there is no need to add missing bits to the integer and fractional parts.

In general, the drawback is that some numbers are not precisely represented with fixed-point arithmetic. As an example, if $m = 4$ and n=4, then the closest representable numbers to 0.7 are 0.6875 ($\langle$0000.1011$\rangle$) and 0.75 ($\langle$0000.1100$\rangle$). As a result, the number needs to be rounded and the deviation might eventually change the control flow of the program. However, we have not detected any false results caused by this in our benchmarks.

3.4.3 Arithmetic Overflow and Underflow

Arithmetic overflow and underflow are frequent sources of bugs in embedded software. ANSI-C, like most programming languages, provides basic data types that have

a bounded range defined by the number of bits allocated to them. Some model checkers (e.g., SMT-CBMC [11], F-Soft [71] and Blast [88]) treat program variables either as unbounded integers or do not generate VCs related to arithmetic overflow, and can consequently produce false results. In our work, we generate VCs related to arithmetic overflow and underflow of bit-vectors following the ANSI-C standard. This requires that, on arithmetic overflow of *unsigned* integer types (e.g., *unsigned int*, *unsigned long int*), the result must be interpreted using modular arithmetic as $r \; mod \; 2^w$, where r is the expression rooted with the operation that caused overflow and w is the width of the resulting type in terms of bits [95]. Hence, the result of this encoding is one greater than the largest value that can be represented by the resulting type. This semantics can be encoded trivially using the background theories of the SMT solvers. For each unsigned integer (sub-)expression, we generate a literal $l_{unsigned_overflow}$ to represent the validity of the unsigned operation and add the following definition:

$$l_{unsigned_overflow} \Leftrightarrow (r - (r \; mod \; 2^w)) < 2^w$$

On the other hand, the ANSI-C standard does not define any behaviour on arithmetic overflow of *signed* types (e.g., *int*, *long int*), and only requires that integer *division-by-zero* must be detected. In addition to *division-by-zero* detection, we consider arithmetic overflow of *signed* types on addition, subtraction, multiplication, division and negation operations by defining boundary conditions. For example, we define a literal $l_{overflow^*_{x,y}}$ that is true iff the multiplication of x and y exceeds LONG_MAX (i.e., $x * y > LONG_MAX$) and another literal $l_{underflow^*_{x,y}}$ that is true iff the multiplication of x and y is below LONG_MIN. We use a literal $l_{res_op^*}$ to denote the validity of the signed multiplication with the following definition:

$$l_{res_op^*} \Leftrightarrow (\neg l_{overflow^*_{x,y}} \wedge \neg l_{underflow^*_{x,y}})$$

The constraints on addition, subtraction, and division are encoded in a similar way. The literal $overflow^{\sim}_{x}$ is *true* if and only if the negation of x is outside the interval given by LONG_MIN and LONG_MAX.

3.4.4 Arrays

Arrays are encoded in a straight-forward manner using the SMT domain theories, and we consider the *WITH* operator and index operator $[\,]$ to be part of the encoding [42, 80]. These operators are mapped directly to the functions *store* and *select* of the array theory presented in Section 2.1.3 respectively. The assignment $a' = a \; WITH \; ([i] := v)$ is encoded as a store operation $a' = store\,(a, i, v)$ while $a[i]$ is simply encoded as a select operation $select\,(a, i)$. The theory of arrays employs the notion of unbounded arrays size, but arrays in software are typically of bounded size. This means that if an index variable

i exceeds the size of an array in a program, the value returned might be undefined or a crash might occur. As an example, consider the code fragment shown in Figure 3.4. In order to check for array bounds violation, we simply keep track of the size of the array and generate for array access (i.e., for both *WITH* and [] operations) a VC that ensures that the value of the index is within the known (and fixed) bounds.

```
1 int i, a[N];
2 ...
3 i=nondet_int();
4 j=nondet_int();
5 ...
6 a[i+j]=2*i;
7 ...
```

FIGURE 3.4: Array out of bounds example.

In order to check for the array bounds in line 6 of Figure 3.4, we create a VC to check the array index $i + j$, which does not require the array theory, as follows:

$$i + j \geq 0 \wedge i + j < N \tag{3.10}$$

Armando et al. [11] also encode programs with arrays using the array theory of the SMT solvers, but they do not generate VCs to check for array bounds violation. The SAT-based version of CBMC generates such VCs but the underlying array representation is fundamentally different. Each array a of size s is replaced by s different scalar variables $a_0, a_2, \ldots, a_{s-1}$ and $a' = store(a, i, v)$ is then represented by the following formula [42, 107]:

$$\bigwedge_{j=0}^{s-1} a'_j = ((i = j) \wedge v) \vee (\neg (i = j) \wedge a_j) \tag{3.11}$$

Similarly, $b = select(a, i)$ is represented as follows:

$$\bigwedge_{j=0}^{s-1} (i = j) \Rightarrow (b = a_j) \tag{3.12}$$

The size of the propositional formulae (3.11) and (3.12) depends on the bit-width of the scalar data types and the size of the arrays occurring in the program, as observed by [11]. In addition, all high-level structure present in the original formula is lost. In contrast, our approach yields more compact VCs and keeps the inherent structure.

3.4.5 Structures and Unions

Structures and unions are encoded using the theory of tuples in SMT and we map update and access operations to the functions *store* and *select* of the theory of tuples presented

in Section 2.1.3. Let w be a structure type, f be a field name of this structure, and v be an expression matching the type of *f*. The expression *store(w,f,v)* returns a tuple that is exactly the same as w except that the value of field f is v; all other tuple elements remain the same. Formally, if *w'=store(w,f,v)* and j is a field name of w, then:

$$w'.j = \begin{cases} v & \text{if } j = f, \\ w.j & \text{if } j \neq f \end{cases} \tag{3.13}$$

In contrast to the situation with arrays, we do not need to generate any VCs, since the field names cannot be computed at run-time, and illegal names would lead to syntactically incorrect programs. We encode unions in a similar way. The difference is that we add an additional field *tag* to indicate the (number of the) field that was used last for writing into the union. This is used to insert the required type-cast operations if any subsequent read access uses a different field. As an illustrative example, consider the fragment of code as shown in Figure 3.5, which contains the union *u_type* and three assertions to check the value of the *u_type* fields.

```
1  union u_type {
2    int i;
3    char ch;
4  };
5  int main() {
6    union u_type u;
7    u.i=1;
8    assert(u.i==1);
9    u.ch='a';
10   assert(u.ch=='a');
11   assert(u.i==97);
12 }
```

FIGURE 3.5: ANSI-C program with union.

In this example, the union *u_type* is modelled as a tuple u with three fields i, *ch* and *tag*, where *tag* indicates the field that was used last for writing into the union u. Note that in our implementation, we identify the fields *tag*, i and *ch* by the numbers 0, 1 and 2 respectively. Note that before we generate C and P as shown in (3.14) and (3.15), we first convert the ASCII character ‘a’ to its respective decimal representation (i.e., 97) in order to avoid using the word-level functions to typecast the character into an integer.

$$C := \left[\begin{array}{l} u_1 = store(store(u_0, 1, 1), 0, 1) \\ \wedge u_2 = store(store(u_1, 2, 97), 0, 2) \end{array} \right] \tag{3.14}$$

$$P := \left[\begin{array}{l} select(u_1, select(u_1, 0)) = 1 \\ \wedge select(u_2, select(u_2, 0)) = 97 \\ \wedge select(u_2, select(u_2, 0)) = 97 \end{array} \right] \tag{3.15}$$

In contrast, the SMT-based BMC approach proposed by Armando et al. [11] does not support unions; Clarke [42, 107] and Kroening [105] encode structs and unions by concatenating all fields into a single bit-vector and extracting them again. This approach, however, might be less scalable because high-level information is lost and therefore, needs to be re-discovered by the SAT or SMT solver (possibly with a substantial performance penalty).

3.4.6 Pointers

In ANSI-C, pointers (and pointer arithmetics) are used as alternative to array indexing: $*(p+i)$ is equal to $a[i]$, if p has been assigned a (see Figure 3.6). The CProver framework removes all pointer dereferences during the unwinding phase and treats pointers as program variables. CBMC's VCG uses the predicate *SAME_OBJECT* to represent that two pointer expressions point to the same memory location or same object. Note that *SAME_OBJECT* is not a safety property, but is mainly used to produce sensible error messages. The VCG generates safety properties that check that *(i)* the pointer offset does not exceed the object bounds (represented by *LOWER_BOUND* and *UPPER_BOUND*) and *(ii)* the pointer is neither `NULL`[2] nor an invalid object (represented by *INVALID_POINTER*). Our approach is similar to the encoding of CBMC into propositional logic, but we use the background theories such as tuples, integer and bit-vector arithmetic while CBMC encodes them by concatenating and extracting the bit-vectors, which operates at the bit-level and is thus less scalable.

We exploit two assumptions about the memory model that is valid in most architectures: no object has address 0 (i.e., $\&a \neq NULL$ and $\&b \neq NULL$) and different objects do not share the same address (i.e., $\&a \neq \&b$). We thus encode pointers into SMT using two fields of a tuple p such that $p.o$ encodes the object the pointer points to, while the $p.i$ encodes an offset within that object. Note that the object can be an array, a struct, or a scalar and that the interpretation of $p.i$ depends on the type of the object: for arrays, it denotes the index, for structs the field, and for scalar it is fixed to zero. Note further that we update the object field $p.o$ dynamically (using the *store* operation of the tuple theory) to accommodate changes of the object that the pointer points to.

Formally, let p_a and p_b be pointer variables pointing to the objects a and b and let η denote the `NULL` pointer encoded as a unique identifier. We encode *SAME_OBJECT* by a literal l_{same_object} with the following definition:

$$l_{same_object} \Leftrightarrow (p_a.o = p_b.o) \wedge (select\,(p_a.o, p_a.i) \neq \eta) \wedge (select\,(p_b.o, p_b.i) \neq \eta) \tag{3.16}$$

A pointer p may point to a set of objects (denoted by O_p) during its lifetime. Whenever p

[2]Note that the ANSI-C guarantees that the NULL pointer compares to the integer zero and can be obtained by converting the integer zero to a pointer type.

is used as array indexing, we check whether p points to some object (in O_p) and if so, we check the upper and lower bounds of p's index. Formally, in order to check the pointer index, we define the upper and lower bound of an object b by b_u and b_l respectively. We then encode the properties *LOWER_BOUND* and *UPPER_BOUND* by creating two literals l_{lower_bound} and l_{upper_bound} with the following definitions:

$$
\begin{aligned}
l_{lower_bound} &\Leftrightarrow ((p_a.o = b \wedge select\,(p_a.o, p_a.i) \neq \eta) \Rightarrow \neg\,(p_a.i < b_l)) \\
l_{upper_bound} &\Leftrightarrow ((p_a.o = b \wedge select\,(p_a.o, p_a.i) \neq \eta) \Rightarrow \neg\,(p_a.i \geq b_u))
\end{aligned} \tag{3.17}
$$

To check invalid pointers, let ν denote an invalid object that is neither `NULL` nor a valid object, i.e., ν is an object that is not any of the existing objects. If p denotes a pointer expression, we encode the property *INVALID_POINTER* by a literal $l_{invalid_pointer}$ with the following definition:

$$
l_{invalid_pointer} \Leftrightarrow (p.o \neq \nu) \wedge (select\,(p.o, p.i) \neq \eta) \tag{3.18}
$$

As example, consider the C program of Figure 3.6 where the pointer p points to the array a as shown in line 3. We build the constraints and properties shown in (3.19) and (3.20) so that the assignment $p{=}a$ in line 3 is converted into a tuple p. The first two conjuncts $p_1 = store\,(p_0, 0, a)$ and $p_2 = store\,(p_1, 1, 0)$ of (3.19) store the object (i.e., array a) and the index 0 at the first two positions of the tuple p.

```
1 int main() {
2   int a[2], i, x, *p;
3   p=a;
4   if (x==0)
5     a[i]=0;
6   else
7     a[i+1]=1;
8   assert(*(p+2)==1); //violated
9 }
```

FIGURE 3.6: C program with pointer to an array.

$$
C := \left[\begin{array}{l}
p_1 = store\,(p_0, 0, a_0) \\
\wedge\ p_2 = store\,(p_1, 1, 0) \wedge\ g_1 = (x_1 = 0) \\
\wedge\ a_1 = store(a_0, i_0, 0) \\
\wedge\ a_2 = a_0 \\
\wedge\ a_3 = store(a_2, 1 + i_0, 1) \\
\wedge\ a_4 = ite(g_1, a_1, a_3) \\
\wedge\ p_3 = store\,(p_2, 1, select\,(p_2, 1) + 2)
\end{array}\right] \tag{3.19}
$$

$$P := \left[\begin{array}{l} i_0 \geq 0 \land i_0 < 2 \\ \land\ 1 + i_0 \geq 0 \land 1 + i_0 < 2 \\ \land\ select\,(p_3, 0) = a_0 \land select\,(p_3, 1) \neq \eta \\ \land\ select\,(select\,(p_3, 0)\,, select\,(p_3, 1)) = 1 \end{array} \right] \tag{3.20}$$

In order to check the property specified in line 8, we first add the value 2 to $p.i$ (i.e., $p_3 = store\,(p_2, 1, select\,(p_2, 1) + 2)$ as shown in the last expression of (3.19)) and then check whether p and a point to the same memory location (as shown in (3.20)). As the value returned by $select(p_3, 1)$ exceeds the size of the object stored in $p_3.o$, (i.e., array a), the *SAME_OBJECT* property is violated and thus the `assert` macro in line 8 fails.

Pointers to structures consisting of n fields with scalar data types are also manipulated like an array with n elements. This means that the CProver framework allows us to encode the structures by using the usual update and access operations. If the structure contains arrays, pointers and scalar data types, then $p.i$ points to the object within the structure only. As an example, Figure 3.7 shows a C program that contains a pointer to a *struct* consisting of two fields (an array a of integer and a *char* variable b). As the *struct* y is declared as global in Figure 3.7 (see lines 1-4), its members must be initialized before performing any operation [95], as shown in the first two lines of (3.21). The assignment $p = \&y$ (see line 7 of Figure 3.7) is encoded by assigning the structure y to the field $p_1.o$ and the value 0 to the field $p_1.i$. The assertions in line 10 and 11 of the C program in Figure 3.7 are simply encoded using the *select* operation of the tuple theory (presented in Section 2.1.3) as shown in (3.22).

```
1  struct x {
2    int a[2];
3    char b;
4  } y;
5  int main(void) {
6    struct x *p;
7    p=&y;
8    p->a[1]=1;
9    p->b='c';
10   assert(p->a[1]==1);
11   assert(p->b=='c'); //ASCII 99
12 }
```

FIGURE 3.7: C program with pointer to a struct.

$$C := \left[\begin{array}{l} y_0.b := 0 \\ \land\ y_1 := store(store(y_0.a, 0, 0), 1, 0) \\ \land\ p_1.o := y \ \land\ p_1.i := 0 \\ \land\ y_2 := store(y_1, a, store(y_1.a, 1, 1)) \\ \land\ y_3 := store(y_2, b, 99) \end{array} \right] \tag{3.21}$$

$$P := \left[\begin{array}{l} select(select(y_3, a), 1) = 1 \\ \wedge\ select(y_3, b) = 99 \end{array} \right] \tag{3.22}$$

3.4.7 Dynamic Memory Allocation

Although dynamic memory allocation is discouraged in embedded software, ESBMC is capable of model checking programs that use it through the ANSI-C functions *malloc* and *free*. We model memory just as an array of bytes and exploit the array theories of SMT solvers to model read and write operations to the memory array on the logic level. ESBMC checks three properties related to dynamic memory allocation; in particular, it checks whether *(i)* the argument to any *malloc*, *free*, or dereferencing operation is a dynamic object (*IS_DYNAMIC_OBJECT*), *(ii)* the argument to any *free* or dereferencing operation is still a valid object (*VALID_OBJECT*), and *(iii)* whether the memory allocated by the *malloc* function is deallocated at the end of an execution (*DEALLOCATED_OBJECT*) [48]. The last check extends CProver framework VCG.

Formally, let p_o be a pointer expression that points to the object o of type t and let m be a memory array of type t and size n, where n represents the number of elements to be allocated. In our encoding, the representation of each dynamic object d_o contains a unique identifier ρ that indicates the object's "serial number" in the sequential order of all dynamically allocated objects (i.e., $0 < \rho \leq k$, where k represents the current number of dynamic objects). Each dynamic object consists of the memory array m, the size in bytes of m, the unique identifier ρ and the location in the execution where m is allocated, which is used for error reporting.

To detect invalid reads/writes, we check whether d_o is a dynamic object and also whether p_o is within the bounds of the memory array. Let i be an integer variable that indicates the position in which the object pointed to by p_o must be stored in the memory array m of size n. We encode *IS_DYNAMIC_OBJECT* as a literal $l_{is_dynamic_object}$ with the following definition:

$$l_{is_dynamic_object} \Leftrightarrow \left(\bigvee_{j=1}^{k} d_o.\rho = j \right) \wedge (0 \leq i < n) \tag{3.23}$$

To check for invalid objects, we add one additional bit field ν to each dynamic object which indicates whether the object is still alive or not. We set ν to *true* when the function *malloc* is called to denote that the object is alive. When the function *free* is called, we update ν to *false* to denote that the object is no longer alive. We then encode *VALID_OBJECT* as a literal l_{valid_object} with the following definition:

$$l_{valid_object} \Leftrightarrow (l_{is_dynamic_object} \Rightarrow d_o.\nu) \tag{3.24}$$

To detect forgotten memory, we check, at the end of the (unrolled) program, for each dynamic object whether it has been deallocated by the function *free*. We can use the existing flag, encoding *DEALLOCATED_OBJECT* as a literal $l_{deallocated_object}$ with the following definition:

$$l_{deallocated_object} \Leftrightarrow (l_{is_dynamic_object} \Rightarrow \neg d_o.\nu) \tag{3.25}$$

Note that the difference between *VALID_OBJECT* and *DEALLOCATED_OBJECT* is the location at which they are checked: *VALID_OBJECT* is checked for each access to a pointer variable, while *DEALLOCATED_OBJECT* is checked only immediately before the (unrolled) program terminates. Note further that both allocation location and size of each dynamic object are immutable whereas the bit field ν is updated when the functions *malloc* and *free* are called.

As an illustrative example, consider the fragment of code as shown in Figure 3.8, which contains two pointers p in line 3 and q in line 4 and allocates three dynamic objects (d_{o1} in line 3, d_{o2} in line 4 and d_{o3} in line 7) of five bytes each. This program contains a typical memory leak since the pointer reassignment in line 5 makes the dynamic object d_{o1} to become an orphan (i.e., $d_{o2}.\nu$ is set to *false* in line 6, but $d_{o1}.\nu$ is still *true*) and as a result the literal $l_{deallocated_object}$ that encodes the deallocation of d_{o1} becomes *false* (i.e., there is a property violation). The constraints and properties of this program are shown in (3.26) and (3.27). The intuitive interpretation of these formulae is that the memory location pointed by p in line 3 (i.e., d_{o1}) cannot be freed because there is no reference to this location, which then results in a memory leak of 5 bytes.

```
1 #include <stdlib.h>
2 void main(){
3   char *p = malloc(5);  // ρ=1
4   char *q = malloc(5);  // ρ=2
5   p = q;
6   free(p);
7   p = malloc(5);        // ρ=3
8   free(p);
9 }
```

FIGURE 3.8: A fragment of an ANSI-C program with dynamic memory allocation.

$$C := \left[\begin{array}{l} d_{o1}.\rho = 1 \wedge d_{o1}.size = 5 \wedge d_{o1}.\nu = true \wedge p = d_{o1} \\ \wedge d_{o2}.\rho = 2 \wedge d_{o2}.size = 5 \wedge d_{o2}.\nu = true \wedge q = d_{o2} \\ \wedge p = d_{o2} \wedge d_{o2}.\nu = false \\ \wedge d_{o3}.\rho = 3 \wedge d_{o3}.size = 5 \wedge d_{o3}.\nu = true \wedge p = d_{o3} \\ \wedge d_{o3}.\nu = false \end{array} \right] \tag{3.26}$$

$$P := \left[\neg d_{o1}.\nu \wedge \neg d_{o2}.\nu \wedge \neg d_{o3}.\nu \right] \tag{3.27}$$

3.5 Experimental Evaluation

The experimental evaluation of the approach presented in this chapter consists of five parts. After describing the setup in Section 3.5.1, we compare in Section 3.5.2, the SMT solvers Boolector, CVC3, and Z3 to identify the most suitable SMT solver for further experiments. In Section 3.5.3 we check the error detection capability of ESBMC over a large set of both correct and buggy ANSI-C programs. In the last two subsections, we evaluate ESBMC's performance relative to that of two other ANSI-C BMC tools. In Section 3.5.4, we compare ESBMC and SMT-CBMC, using SMT-CBMC's own benchmark suite, while we compare ESBMC and CBMC in the final Section 3.5.5, using a variety of programs, including embedded software used in telecommunications, control systems, and medical devices. Section 3.6 contains the experimental results of applying ESBMC and CBMC to the verification of a commercial embedded software. The purpose of this section is to evaluate both tools ESBMC and CBMC using large embedded software industrial applications.

3.5.1 Experimental Setup

We used benchmarks from a variety of sources to evaluate ESBMC's precision and performance, which include embedded systems benchmark suites and applications as well as other testsuites and applications, such as the SAT solver PicoSAT [23], the open-source applications *flex* [153] and *git-remote* [132], and a flasher manager application [175]. We also extracted one particular application from the CBMC manual [42] that implements the multiplication of two numbers using bit-level operations.

The PowerStone [159] suite contains graphics applications, image decompression, paging communication protocols, engine control applications and group three fax decode. The SNU-RT [116] suite consists of matrix and signal processing functions such as matrix multiplication and decomposition, quadratic equations solving, insertion sort algorithm, cyclic redundancy check, fast Fourier transform, LMS adaptive signal enhancement, and JPEG encoding. We use the non-deterministic version of these benchmarks where all inputs are replaced by non-deterministic values. We also a cubic equation solver from the MiBench [2] suite. The HLS suite [86] contains programs that implement the encoder and decoder of the adaptive differential pulse code modulation (ADPCM).

The NECLA [157] and VERISEC [110] benchmarks are not specifically related to embedded software, but they allow us to check ESBMC's error-detection capability easily since they provide ANSI-C programs with and without known bugs. Here, we use the suffix "-bad" to denote the subset with seeded errors, and "-ok" to the denote the supposedly correct ("golden") versions.[3] The programs make use of dynamic memory allocation,

[3] The detailed results shown in Appendix B also show which programs are "bad" and which are "ok".

interprocedural dataflow, aliasing, pointers typecast and string manipulation. In addition, we used some programs from the well-known Siemens [142] test suite, including pattern matching and string processing, statistics, and aerospace applications. The EUREKA [11] benchmarks finally contain programs that allow us to assess the scalability of the model checking tools on problems of increasing complexity [11].

Unless stated otherwise, all experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 4 GB of RAM running Linux OS. For all benchmarks, the time limit has been set to 3600 seconds for each individual property. All times given are wall clock time in seconds as measured by the unix *time* command.

3.5.2 Comparison of SMT solvers

As a first step, we compared to which extent the SMT solvers support the domain theories that are required for SMT-based BMC of ANSI-C programs. For this purpose, we analyzed the SMT solvers Boolector (V1.4), CVC3 (V2.2), and Z3 (V2.11). In the *theory of linear and non-linear arithmetic*, CVC3 and Z3 do not support the remainder operator, but they allow us to use axioms to define it. Currently, Boolector does not support the theory of linear and non-linear arithmetic at all. In the *theory of bit-vectors*, CVC3 does not support the division and remainder operators for bit-vectors representing signed and unsigned integers. However, in all cases, axioms can be used in order to define the missing operators. Boolector and Z3 support all word-level, bit-level, relational, arithmetic functions over unsigned and signed bit-vectors. In the theories of *arrays* and *tuples*, the verification problems only involve selecting and storing elements from/into arrays and tuples, respectively, and both domains thus comprise only two operations. These operations are fully supported by CVC3 and Z3; Boolector supports only the theory of arrays but not that of tuples.

We then used 15 ANSI-C programs to compare the performance of Boolector, CVC3, and Z3 as ESBMC back-ends. The programs 1-8 allow us to assess the scalability of the model checking tools on problems of increasing complexity [11] and the programs 9-15 contain typical ANSI-C constructs found in embedded software, i.e., they contain linear and non-linear arithmetic and make heavy use of bit operations.

	Program	L	B	P	CVC3 (v2.2) Solver	CVC3 (v2.2) Total	Boolector (v1.4) Solver	Boolector (v1.4) Total	Z3 (v2.11) Solver	Z3 (v2.11) Total
1	EUREKA.BubbleSort	43	35	17	14 (3)	17 (5)	<1 (<1)	**2** **(2)**	<1 (<1)	**2** (3)
		43	70	17	M_b (16)	M_b (33)	3 (1)	**16** (17)	3 (1)	**16** (17)
		43	140	17	M_b (M_b)	M_b (M_b)	85 (53)	282 (311)	65 (11)	**265** (269)
2	EUREKA.SelectionSort	34	35	17	17 (2)	18 (3)	<1 (<1)	**1** **(1)**	<1 (<1)	**1** **(1)**
		34	70	17	M_b (8)	M_b (17)	1 (<1)	**9** (10)	1 (1)	**9** (11)
		34	140	17	M_b (42)	M_b (209)	10 (3)	**161** (171)	12 (6)	165 (173)
3	EUREKA.BellmanFord	49	20	33	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
4	EUREKA.Prim	79	8	30	<1 (1)	5 (2)	<1 (<1)	<**1** **(<1)**	<1 (<1)	<**1** **(<1)**
5	EUREKA.StrCmp	14	1000	6	4 (444)	**11** (454)	192 (248)	195 (257)	32 (37)	35 (46)
6	EUREKA.SumArray	12	1000	7	<1 (106)	**1** (107)	<1 (<1)	**1** **(1)**	9 (<1)	10 **(1)**
7	EUREKA.MinMax	19	1000	9	T_b (M_b)	T_b (M_b)	38 (2)	42 (7)	2 (1)	**6** (7)
8	SNU-RT.InsertionSort	34	35	17	2 (3)	4 (5)	<1 (<1)	**3** **(3)**	<1 (<1)	**3** **(3)**
		34	70	17	3 (11)	14 (24)	4 (<1)	15 **(13)**	2 (1)	**12** (14)
		34	140	17	21 (67)	**194** (283)	193 (3)	350 (219)	42 (7)	212 (222)
9	SNU-RT.Fibonacci	40	30	4	<1 (<1)	39 (38)	<1 (<1)	39 (38)	<1 (<1)	39 (38)
10	SNU-RT.bs	95	15	7	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
11	SNU-RT.lms	258	202	23	97 (17)	**225** (324)	<1 (<1)	303 (307)	3 (<1)	306 (307)
12	MiBench.Cubic	66	5	5	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)
13	CBMC.BitWise	18	8	1	3 (6)	**3** (6)	7 (8)	7 (8)	30 (26)	30 (26)
14	HLS.adpcm_encode	149	200	12	<1 (21)	6 (26)	<1 (<1)	6 (6)	<1 (<1)	6 (6)
15	HLS.adpcm_decode	111	200	10	<1 (24)	3 (27)	<1 (<1)	3 (3)	<1 (<1)	3 (3)

Table 3.2: Results of the comparison between CVC3, Boolector and Z3. Time-outs are represented with T in the Time column; Examples that exceed available memory are represented with M in the Time column. The subscript b indicates that the error occurred in the back-end.

Table 3.2 shows the results of the comparison. Here, L is the number of lines of code, B the unwinding bound, and P the number of properties verified, for each ANSI-C program. We checked for all language-specific safety properties (as described in the previous sections) as well as user-specified properties. For each solver, we provide the total time (in seconds) to simultaneously check all properties of each program, using the specified unwinding bound, as well as the solver time itself. The difference between both times is spent in the ESBMC front-end. In addition, we provide (in brackets) the timings using the SMT-LIB interface instead of the native API of the solver.[4] The fastest time for each program is shown in bold. We also indicate whether ESBMC fails during the verification process, either due to a time out (T) or due to memory overflow (M). In this set of experiments, all failures occurred in the back-end (i.e., solver), which is indicated by the subscript b.

As we can see in Table 3.2, if we use the native API of the solvers, Z3 usually runs slightly faster than Boolector and CVC3; however, both CVC3 and Boolector are faster for some programs. Generally the differences between the solvers (in particular between Boolector and Z3) are small, although CVC3 fails for some examples. If we use the SMT-LIB interface, the situation changes, and Boolector runs slightly faster than Z3 and CVC3. However, similar to case of the native API, it is not always the fastest solver; again, the differences are generally small, and even smaller than when using the native API.

Generally, the native API is slightly faster than the SMT-LIB interface, although the difference is small as well; this happens because in the SMT-LIB interface, we have to write/read the resulting SMT formula to/from a file in the disk in order to interact with the SMT solver, which is extremely slower than accessing the SMT solver directly through the native API. However, there are a few notable exceptions where the SMT-LIB interface is slightly faster than the native API. Using the SMT-LIB interface, CVC3 scales better for *BubbleSort* and *SelectionSort*, but slows down substantially for *StrCmp* and *SumArray*. We manually inspected the respective VCs and found that their structure is essentially the same. We conclude that the SMT-LIB interface of CVC3 lacks some optimization during the preprocessing. Similarly, Boolector speeds up for *InsertionSort* using the SMT-LIB API, but the structure of the VCs using both APIs is also the same; similarly, we conclude that the SMT-LIB interface enables some optimization during the preprocessing.

We decided to continue the evaluation with Z3 and Boolector using both the native and SMT-LIB APIs since CVC3 does not scale so well and fails to check three benchmarks *BubbleSort*, *SelectionSort* and *MinMax*.

[4]See Chapter 5 for a detailed description of the different solver integrations.

					Time		Properties		Errors		
	Testsuite	#N	ΣL	ΣP	Solver	Total	Passed	Violated	$\#N_e$	true	false
1	EUREKA	7	787	420	182	543	420	0	-	-	-
2	NECLA-ok	30	891	254	98	172	212	42	2	3	0
	NECLA-bad	10	342	112	37	47	87	25	10	25	0
3	POWERSTONE	9	2857	2031	728	816	2019	12	1	12	0
4	SNU-RT	20	3320	828	15	570	799	29	4	29	0
5	VERISEC-ok	80	4521	2114	128	211	2094	15	9	15	0
	VERISEC-bad	83	4569	2024	127	226	1808	216	83	216	0
6	WCET	10	3430	726	7	73	722	4	2	3	1

Table 3.3: Results of the error-detection capability of ESBMC.

3.5.3 Error-Detection Capability

We now analyze to which extent ESBMC is able to handle and detect errors in standard ANSI-C benchmarks. Table 3.3 summarizes the results. Here, N is the number of programs in the benchmark suite, while ΣL and ΣP give its total size (in lines of code) and the total number of properties checked, respectively. The table again shows both the solver and total verification time. In the last three columns, N_e is the number of programs in which ESBMC has detected violations of safety properties and user-specified assertions, "true" reports the number of property violations that correspond to true, confirmed faults, "false" reports the number of false negatives produced by ESBMC. The Appendix B gives the complete results.

The EUREKA suite only contains correct programs and ESBMC is able to verify all properties without producing any false negative. In the NECLA and VERISEC suites, ESBMC is able to detect errors related to buffer overflow, aliasing, dynamic memory allocation, and string manipulation; in particular, it detects all seeded errors in the versions NECLA-bad and VERISEC-bad. Moreover, ESBMC could verify two programs that were originally in NECLA-bad, but did not contain any seeded errors; the benchmark creators confirmed that these programs were misclassified and subsequently changed the error seeding [97].

Surprisingly, ESBMC also detects errors in the supposedly correct golden versions. In NECLA-ok, ESBMC finds three property violations in two programs, which have been confirmed as true faults by the benchmark creators [97]. The first is an array bounds violation, caused by an indexing expression $x\%32$ that can become negative for negative inputs x. The other two are also related to array bounds violations, but are caused by repeated in-place updates of a buffer using the *strcat*-function, which also appends a new NULL-character at the end of the new string formed by the concatenation of both arguments; this NULL-character then causes the violation in the last iteration of the

loop. In VERISEC-ok, ESBMC finds 15 property violations in nine programs, which have also been confirmed by the benchmark creators [36]. All violations are related to arithmetic overflow on the typecast operation caused by assignments of the form $c=i$, where c is declared as a *char* and i as an *int*.

In the WCET test suite, ESBMC finds four property violations in two programs, which we inspected manually. Two violations point to possible overflows that stem from assignments between incompatible datatypes (e.g., *long int* vs. *int*), which are indeed errors; a further violation points to a potential division by zero error, which is very unlikely to be uncovered by testing, as it requires an entire array to be randomly initialized with zeroes. The final property indicates an arithmetic overflow in an expression *StopTime-StartTime*, but this is a false negative, since both variables are guaranteed to be positive at runtime, and moreover, *StopTime* is always larger than *StartTime*. This false negative can be suppressed by adding an assumption on the return values to the *ttime*-function that is used to compute both variables. Finally, ESBMC finds array bounds violations and overflows in arithmetic expressions in four of the SNU-RT benchmarks and invalid pointers in one of the PowerStone benchmarks; we confirmed by inspection that these are indeed faults.

3.5.4 Comparison to SMT-CBMC

This subsection describes the evaluation of ESBMC against another SMT-based BMC developed by Armando et al. [11]. For the evaluation, we took the official benchmark of the tool [148], because it does not support some of the ANSI-C constructs commonly found in embedded software (e.g., bit operations, fixed-point arithmetic, pointer arithmetic); Table 3.4 summarizes the results. The timings in brackets again refer to ESBMC's SMT-LIB interface; we do not report for SMT-CBMC because it does not output the formula in the SMT-LIB format (SMT-CBMC uses the native API of the solver only). Note that results given for ESBMC differ from those in Table 3.2: since SMT-CBMC does not generate any checks for safety properties we used both systems only to check the single user-specified property. SMT-CBMC has been invoked by setting manually the file name and the unwinding bound (i.e., SMT-CBMC `-file` *Module* `-bound` *B*). Furthermore, we compared SMT-CBMC with its default solver (i.e, CVC3 2.2) against ESBMC using both its default solver (i.e., Z3 2.11) as well as CVC3 2.2.

If CVC3 is used as the SMT solver, both tools run out of memory and thus fail to analyze *BubbleSort* for large B (B=140). SMT-CBMC runs out of time when analyzing the program *SelectionSort* and *StrCmp* while ESBMC runs out of time for the program *MinMax*. ESBMC outperforms SMT-CBMC by a factor of 6-90 for those benchmarks that do not fail. However, if Z3 is used as solver for ESBMC, the difference between both tools becomes more noticeable and ESBMC generally outperforms SMT-CBMC by a factor of 10-200. We can conclude that SMT-CBMC has limitations not only in the

	Module	L	B	P	ESBMC (Z3)		ESBMC (CVC3)		SMT-CBMC
					Solver	Total	Solver	Total	Total
1	EUREKA.BubbleSort	43	35	1	<1 (<1)	2 (2)	15 (3)	16 (3)	100
		43	60	1	2 (<1)	7 (8)	128 (9)	134 (16)	304
		43	70	1	3 (1)	13 (15)	M_b (16)	M_b (30)	407
		43	140	1	68 (11)	259 (265)	M_b (M_b)	M_b (M_b)	M
		43	180	1	M_f (M_f)	M_f (M_f)	M_f (M_f)	M_f (M_f)	M
2	EUREKA.SelectionSort	34	35	1	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	T
		34	70	1	<1 (<1)	8 (9)	<1 (7)	7.6 (15)	T
		34	140	1	10 (4)	157 (162)	2 (34)	160 (193)	T
		34	170	1	18 (6)	336 (344)	2 (53)	323 (392)	T
		34	180	1	M_f (M_f)	M_f (M_f)	M_f (M_f)	M_f (M_f)	T
3	EUREKA.BellmanFord	49	20	1	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	43
4	EUREKA.Prim	79	8	1	<1 (<1)	<1 (<1)	<1 (<1)	<1 (<1)	96
5	EUREKA.StrCmp	14	1000	1	25 (30)	27 (38)	3 (253)	7 (261)	T
6	EUREKA.SumArray	12	1000	1	9 (<1)	25 (<1)	<1 (108)	<1 (108)	98
7	EUREKA.MinMax	19	1000	1	2 (1)	6 (6)	T_b (M_b)	T_b (M_b)	65

TABLE 3.4: Results of the comparison between ESBMC and SMT-CBMC [11].

verification time (due to the lack of simplification based on high-level information; see Section 5.3 for more details), but also in the encodings of important ANSI-C constructs used in embedded software.

3.5.5 Comparison to CBMC

CBMC [42] is one of the most widely used BMC tools for ANSI-C. It has recently been extended by an SMT backend [105], and in our comparison we tried to use the SMT solvers Z3 and Boolector (by invoking `--z3` or `--boolector`) for evaluating both tools CBMC and ESBMC. However, the SMT-based CBMC version failed to check all benchmarks reported in Table 3.5 due to problems in the SMT back-end. Consequently, we compare our approach only against the SAT-based CBMC version, which is able to support most of the benchmarks from Table 3.5; in particular, we compared CBMC v3.8 and ESBMC v1.15. We invoked both tools by manually setting the file name, the unwinding bound, the checks for array bounds, pointer safety, division by zero, and arithmetic over- and underflow.[5] Table 3.5 reports the results in the usual format.

As we can see in Table 3.5, SAT-based CBMC is not able to check the module *pocsag* due to memory limitations; it times out in five cases and fails in four cases due to errors in the front-end, and in another five cases due to errors in the back-end. ESBMC runs out of time to check the modules *qurt* and *ludcmp*, but it is able to check seven (of

[5] The tools where invoked as follows: `cbmc file --unwind` *B* `--bounds-check --div-by-zero-check --pointer-check --overflow-check --string-abstraction` and `esbmc file --unwind` *B* `--overflow-check --string-abstraction`.

eight) properties of the module *qurt* and fifteen additional benchmarks in comparison to SAT-based CBMC. Both CBMC and ESBMC find errors in the SNU-RT (as confirmed in Section 3.5.3). However, ESBMC finds additional confirmed errors (see Section 3.5.3 again) in the WCET, SNU-RT, and PowerStone benchmarks, while CBMC produces false negatives or fails.

In the case of *print_tokens2*, ESBMC runs out of memory if we try to increase the unwinding bound to 82, but if we restrict the verification to the function *get_token*, it finds an array-bounds violation in the golden version. We extracted the counterexample provided by ESBMC and used it to confirm that this is a true fault. ESBMC also finds additional errors in *flasher_manager* (violation of a user-specified assertion) and *adpcm_encode* (array-bounds violation) applications. Moreover, SAT-based CBMC also produces false negatives for the golden version of the programs *ex30* and *ex33* by reporting non-existing bugs related to dynamic object upper bounds and invalid pointers. We can also see that ESBMC not only has a better precision than SAT-based CBMC, but it also runs slightly faster than the SAT-based CBMC in those benchmarks that it does not fail. The results in Table 3.5 thus allow us to conclude that ESBMC improves substantially precision and scales significantly better than CBMC for problems that involve tight interplay between non-linear arithmetic, bit operations, pointers and array manipulations, which are typical for embedded systems software.

3.6 Industrial Case Study

In order to further evaluate ESBMC's performance relative to CBMC, we analyzed the embedded software used in a commercial product from NXP semiconductors [141], a set-top box that is used in high definition internet protocol (IP) and hybrid digital TV applications. The embedded software of this platform relies on the Linux operating system and makes use of different applications such as:

1. *LinuxDVB* that is responsible for controlling the front-end, tuners and multiplexers [6].
2. *DirectFB* that provides graphics applications and input device handling [5].
3. *ALSA* that is used to control the audio applications [4].

This platform contains two embedded processors that exchange data via an inter-process communication (IPC) mechanism using socket (which thus allows the communication between the two processors).

					SAT-based CBMC (v3.8) [42]					ESBMC (v1.15)				
					Time		Properties			Time		Properties		
	Module	L	B	P	Solver	Total	Passed	Violated	Fail	Solver	Total	Passed	Violated	Fail
1	Siemens.print_tokens2	510	81^*	135	<1	<1	135	0	0	<1 (<1)	<1 (<1)	135	0	0
	(get_token)	51	82	76	T_b	T_b	0	0	135	29 (35)	**60** (65)	134	1	0
2	Siemens.replace	564	1^*	199	$\dagger^f$	$\dagger^f$	-	-	-	<1 (<1)	**<1 (<1)**	199	0	0
3	Siemens.tot_info	406	30^*	73	$\dagger^f$	$\dagger^f$	-	-	-	32 (3)	98 **(79)**	73	0	0
4	Siemens.tcas	173	4	38	<1	**<1**	38	0	0	1 (<1)	2 (1)	38	0	0
5	Siemens.space	9125	126^*	2016	<1	4	2016	0	0	<1 (<1)	**3 (3)**	2016	0	0
6	WCET.statistics	157	∞	29	$\dagger^f$	$\dagger^f$	-	-	-	1 (<1)	**53 (53)**	27	2	0
7	WCET.statemate	1273	3	6	<1	<1	6	0	0	<1 (<1)	<1 (<1)	6	0	0
8	SNU-RT.crc_new	125	∞	13	<1	**6**	12	1	0	<1 (<1)	8 (8)	12	1	0
9	SNU-RT.fft1k_new	158	∞	39	$\dagger^b$	$\dagger^b$	35	0	4	<1 (1)	**56** (57)	39	0	0
10	SNU-RT.fibcall_new	83	50^*	2	<1	<1	1	1	0	<1 (<1)	<1 (<1)	1	1	0
11	SNU-RT.fir_new	316	∞	25	5	6	25	0	0	<1 (<1)	**2 (2)**	25	0	0
12	SNU-RT.insertsort_new	94	13	20	$\dagger^b$	$\dagger^b$	0	0	20	8 (<1)	**8 (2)**	14	6	0
13	SNU-RT.lms_new	256	∞	35	$\dagger^b$	$\dagger^b$	29	0	6	3 (<1)	**24 (24)**	35	0	0
14	SNU-RT.ludcmp_new	142	∞	79	T_b	T_b	84	0	4	T_b (T_b)	T_b (T_b)	84	0	4
15	SNU-RT.qurt_new	159	∞	8	T_b	T_b	2	0	6	T_b (T_b)	T_b (T_b)	7	0	1
16	PowerStone.bcnt	83	17	153	2	3	153	0	0	2 (2)	**2 (2)**	153	0	0
17	PowerStone.blit	95	1	133	<1	<1	133	0	0	<1 (<1)	<1 (<1)	129	4	0
18	PowerStone.pocsag	521	42	187	M_f	M_f	-	-	-	4 (<30)	**22** (48)	186	1	0
19	NECLA.ex30	45	101	16	<1	**2**	12	4	0	<1 (<1)	3 (3)	16	0	0
20	NECLA.ex33	35	100	13	<1	<1	6	7	0	<1 (<1)	<1 (<1)	13	0	0
21	picosat	8160	23^*	3142	T_f	T_f	-	-	-	27 ($\dagger^b$)	**79** ($\dagger^b$)	3142	0	0
22	flex	14192	2^*	10002	$\dagger^f$	$\dagger^f$	-	-	-	3492 ($\dagger^b$)	**3526** ($\dagger^b$)	10002	0	0
23	git-remote-gitkrb5	6288	5^*	174	$\dagger^b$	$\dagger^b$	0	0	174	196 ($\dagger^b$)	**225** ($\dagger^b$)	174	0	0
24	flasher_manager	521	21	26	2	**4**	26	0	0	25 (22)	29 (27)	25	1	0
25	HLS.adpcm_encode	150	100	25	T_b	T_b	0	0	25	<1 (<1)	**6 (6)**	24	1	0

Table 3.5: Results of the comparison between CBMC and ESBMC. Internal errors in the respective tool are represented with † in the Time column. The subscripts f and b indicate whether the errors occurred in the front-end or back-end, respectively. The superscript $*$ on the unwinding bound indicates that it is not large enough to prove or falsify the properties.

We analyzed the following embedded applications:

1. *exStbKey:* This application checks the DirectFB key codes that are returned when the remote control or front panel keys are pressed.

2. *exStbHDMI:* This application is used to set various capabilities of the HDMI device (e.g., audio rate, video mode) and to read various statuses of the HDMI device (e.g., sink type, hotplug status).

3. *exStbLED:* This application is responsible for setting the front panel LED display; and uses raw keyboard input from the UART to control what is displayed on the front panel LED display.

4. *exStbHwAcc:* This application demonstrates the advantages that can be gained by using the graphics hardware acceleration that is available on the set-top box.

5. *exStbResolution:* This application is responsible for modifying the framebuffer dimensions and upscaling by setting framebuffer to be accessed and updating the width and height of the framebuffer.

6. *exStbFb:* This application is used to decode image files and display them in a framebuffer or on a video layer.

7. *exStbCc:* This application outputs a test closed caption stream.

8. *exStbDemo:* This application is used to demonstrate a multitude of system features in an integrated system. It includes support for DVB reception, channel change, installation, programme information, recording and playback, IP reception and playback (both unicast and multicast formats), media file playback (elementary streams and transport streams), image decoding and display manipulation.

As we did in Section 3.5.5, we compare our approach only against the SAT-based CBMC version, which is able to support most of the benchmarks from Table 3.6; in particular, we again compared CBMC v3.8 and ESBMC v1.15. We also invoked both tools by manually setting the file name, the unwinding bound, the checks for array bounds, pointer safety, division by zero, and arithmetic over- and underflow, as before. Table 3.6 reports the results in the usual format.

Both SAT-based CBMC and ESBMC were able to find a bug in the application *exStb-HwAcc*, which is related to an arithmetic overflow on typecast. In a given part of the program *exStbHwAcc*, there is a typecast operation of the form *(int32_t)(finfo.smem_len)*, which converts the field *smem_len* of type unsigned integer into a signed integer; and this is thus considered to be an overflow. ESBMC also found two bugs in the application *exStbCc*, which are related to arithmetic overflow on addition. In this program, we have the program statement *offset[0] += ret* guarded by an *if* condition, but inside an

infinite loop. Therefore, successive additions of the variable *ret* to the array *offset* can lead to an arithmetic overflow.

We are able to model check the application *exStbDemo* up to the bound 16, and we do not find any property violation. If we increase the unwinding bound further to search deeper on the state space, we are unable to model check *exStbDemo* due to memory limitation and time out resp. as shown in lines 8.1 (running on a machine with 4 GB of RAM) and 8.2 (running on a machine with 28 GB of RAM) of Table 3.6. If we verify the application *exStbDemo* function-by-function, we can thus go deeper into the system and explore more exhaustively the state space. However, ESBMC provides false negatives related to pointer safety since we assume that the function parameters are unconstrained (see functions *readLine* (8.3), *getCommand* (8.4) and *main_Thread*) (8.15). From this set of experiments, we can conclude that the size of the programs that state-of-the-art bounded model checkers can cope with is still restricted (even if we define only small parts of the program to be verified).

3.7 Related Work

There has been work in the verification of low-level (assembly language) programs for embedded systems. Thiry and Claesen [170] apply a model checking algorithm based on binary decision diagrams (BDDs) using the SMV model checker [33] to verify a mouse controller. In this work, however, the authors use the computational tree logic (CTL) to model and verify the embedded software. Thiry and Claesen are able to find inconsistencies between the assembly code and flow chart specifications of the mouse controller. The drawback of this approach is that it is limited to complexity problems in the symbolic state space representation and manipulation using BDDs [25].

In another work, Balakrishnan and Tahar extend the BDD-based model checking algorithm to support the more general multiway decision graph (MDG) to avoid some BDD-size blow-up [15]. The main idea behind MDG is to represent the model at higher abstract levels using a subset of first order logic (FOL) and then make use of the automation offered by BDDs-based tools. Balakrishnan and Tahar also verify the mouse controller case study of [33]. The authors report that with their approach they can also find inconsistencies between the specification and the code in few seconds. This approach, however, is applied to verify one small embedded application and consequently does not demonstrate the verification of real-world embedded software.

					SAT-based CBMC (v3.8) [42]					ESBMC (v1.15)				
					Time		Properties			Time		Properties		
	Module	L	B	P	Solver	Total	Passed	Violated	Fail	Solver	Total	Passed	Violated	Fail
1	exStbKey	558	4	33	<1	4	33	0	0	<1 (<1)	**1 (1)**	33	0	0
2	exStbHDMI	1508	15^*	138	500	706	138	0	0	316 ($\dagger_b$)	**429** ($\dagger_b$)	138	0	0
3	exStbLED	430	50^*	102	72	122	102	0	0	48 (68)	**80** (79)	102	0	0
4	exStbHwAcc	1432	3	239	2	6	238	1	0	<1 ($\dagger_b$)	**1** ($\dagger_b$)	238	1	0
5	exStbResolution	353	50	79	$\dagger_b$	$\dagger_b$	0	0	70	26 (59)	**59** (61)	70	0	0
6	exStbFb	689	10	218	484	825	167	0	0	52 ($\dagger_b$)	**101** ($\dagger_b$)	167	0	0
7	exStbCc	331	3	21	<1	3	19	2	0	<1 (<1)	<**1** (<1)	19	2	0
8	exStbDemo	14841	16^*	471	$\dagger_f$	$\dagger_f$	-	-	-	<1 (<1)	6 (7)	471	0	0
	exStbDemo [4 GB]	14841	17	471	$\dagger_f$	$\dagger_f$	-	-	-	M$_f$ (M$_f$)	M$_f$ (M$_f$)	-	-	-
	exStbDemo [28 GB]	14841	17	471	$\dagger_f$	$\dagger_f$	-	-	-	T$_f$ (T$_f$)	T$_f$ (T$_f$)	-	-	-
8.1	threadRename	6	17	0	<1	3	0	0	0	<1 (<1)	3 (3)	0	0	0
8.2	fileExists	19	17	0	<1	3	0	0	0	<1 (<1)	3 (3)	0	0	0
8.3	readLine	27	17	11	<1	3	10	1	0	<1 (<1)	3 (3)	10	1	0
8.4	getCommand	269	17	61	<1	6	60	1	0	<1 (<1)	**3** (3)	60	1	0
8.5	powerDown	9	17	0	<1	2	0	0	0	<1 (<1)	2 (2)	0	0	0
8.6	digitStart	12	17	0	<1	2	0	0	0	<1 (<1)	2 (2)	0	0	0
8.7	digitAdd	34	17	2	<1	2	2	0	0	<1 (<1)	2 (2)	2	0	0
8.8	checkEndOfPvrStream	32	13	13	<1	2	13	0	0	<1 (<1)	2 (2)	13	0	0
8.9	checkEndOfMediaStream	28	1	1	<1	2	1	0	0	<1 (<1)	2 (2)	1	0	0
8.10	commandLoop	545	17	53	M$_f$	M$_f$	-	-	-	M$_f$ (M$_f$)	M$_f$ (M$_f$)	-	-	-
8.11	checkCommandParams	238	17	269	T$_b$	T$_b$	0	0	269	T$_b$ (T$_b$)	T$_b$ (T$_b$)	0	0	269
8.12	signal_handler	13	17	0	<1	2	0	0	0	<1 (<1)	2 (2)	0	0	0
8.13	setupFBResolution	29	17	0	<1	2	0	0	0	<1 (<1)	2 (2)	0	0	0
8.14	setupFramebuffers	115	17	8	<1	3	8	0	0	<1 (<1)	3 (3)	8	0	0
8.15	main_Thread	68	17	4	T$_f$	T$_f$	-	-	-	<1 (<1)	**4** (4)	3	1	0
8.16	set_to_raw	8	17	0	<1	3	0	0	0	<1 (<1)	3 (3)	0	0	0
8.17	set_to_buffered	8	17	0	<1	2	0	0	0	<1 (<1)	2 (2)	0	0	0

TABLE 3.6: Results of the comparison between CBMC and ESBMC on a industrial case study.

Lettnin et al. [114] describe a semiformal verification methodology that adopts simulation and formal verification. This solution uses the frontend of BLAST tool [88] to convert a C program to a CFG and uses the SymC model checker [83] to verify the design properties. Lettnin et al. apply this methodology in a case study to verify the locking and unlocking rules of a driver. This approach, however, faces memory overflow problems due to the BDD-based model checking algorithm and consequently the authors have to set a threshold on the number of states during the static verification. Lettnin et al. [115] extends [114] to combine assertion-based verification and symbolic simulation for the verification of embedded software with hardware dependencies. However, their approach does not produce counter-examples and therefore becomes hard to debug the code in case of a failing property.

SMT-based BMC is gaining popularity in the formal verification community due to the advent of sophisticated SMT solvers built over efficient SAT solvers [20, 31, 57]. Previous work related to SMT-based BMC [71, 181, 11] combined decision procedures for the theories of uninterpreted functions, arrays and linear arithmetic only, but did not encode key constructs of the ANSI-C programming language such as bit operations, fixed-point arithmetic and pointers. Ganai and Gupta describe a verification framework for BMC which extracts high-level design information from an extended finite state machine (EFSM) and applies several techniques to simplify the BMC problem [71, 72]. However, the authors flatten structures and arrays into scalar variables in such a way that they use only the theory of integer and real arithmetic in order to solve the verification problems that come out in BMC.

Armando et al. also propose a BMC approach using SMT solvers for C programs [11]. However, they only make use of linear arithmetic (i.e., addition and multiplication by constants), arrays, records and bit-vectors in order to solve the VCs. As a consequence, their SMT-CBMC prototype does not address important constructs of the ANSI-C programming language such as non-linear arithmetic and bit-shift operations. Kroening also encodes the VCs generated by the front-end of CBMC by using the bit-vector arithmetic and does not exploit other background theories of the SMT solvers to improve scalability [105]. Donaldson et al. present an approach to compute invariants in BMC of software by means of k-induction [63]. Their method, however, is highly customized for checking assertions representing DMA operations in the Cell processor, which requires only a small number of loop iterations and thus allows k-induction to work well with a small value of k. Xu proposes the use of SMT-based BMC to verify real-time systems by using TCTL to specify the properties [181]. The author considers an informal specification (written in English) of the real-time system and then models the variables using integers and reals and represents the clock constraints using linear arithmetic expressions.

De Moura et al. present a bounded model checker that combines propositional SAT solvers with domain-specific theorem provers over infinite domains [59]. Differently

from other related work, the authors abstract the Boolean formula and then apply a lazy approach to refine it in an incremental way. This approach is applied to verify timed automata and RTL level descriptions. Jackson et al. [98] discharge several VCs from programs written in the Spark language to the SMT solvers CVC3 and Yices as well as to the theorem prover Simplify. The idea of this work is to replace the Praxis prover by CVC3, Yices and Simplify in order to generate counter-example witnesses to VCs that are not valid. In [99], Jackson and Passmore extend [98] by implementing a tool to automatically discharge VCs using SMT solvers. The authors observed significant performance improvement of the SMT solvers if compared to the Praxis prover. Jackson and Passmore, however, focus on translating VCs into SMT from programs written in the SPARK language (which is a subset of the Ada language) instead of ANSI-C programs.

Recently, a number of static checkers have been developed in order to trade off scalability and precision. Calysto is an automatic static checker that is able to verify VCs related to arithmetic overflow, null-pointer dereferences and assertions specified by the user [13]. The VCs are passed to the SMT solver SPEAR which supports boolean logic and bit-vector arithmetic and is highly customized for the VCs generated by Calysto. However, Calysto does not support floating-point operations and unsoundly approximates loops by unrolling them only once. As a consequence, soundness is relinquished for performance. Saturn is another automatic static checker that scales to larger systems, but with the drawback of losing precision by supporting only the most common integer operators and performing at most two unwindings of each loop [180]. In contrast to [13, 180], the extended static checker for Java (ESC/JAVA) is a semi-automatic verification tool, which requires the programmer to supply loop, function, and class invariants and thus limits its acceptance in practice [69]. In addition, ESC/Java employs the Simplify theorem prover [62] to verify user-supplied invariants and thus important constructs of the programming language (e.g., bitwise operation) are often encoded imprecisely using axioms and uninterpreted functions.

3.8 Conclusions

In this chapter, we have investigated SMT-based verification of ANSI-C programs, with a focus on embedded software. In this sense, we have described a new set of encodings that allow us to reason accurately about bit operations, unions, fixed-point arithmetic, pointers and pointer arithmetic and implemented it in the ESBMC tool. As far as we are aware, no encoding into the SMT theories existed that can reliably handle full ANSI-C. With these encodings we have successfully achieved the first objective stated in Section 1.2.

Moreover, our experiments constitute, to the best of our knowledge, the first substantial evaluation of SMT-based BMC on industrial applications. The results show that

ESBMC outperforms CBMC [42] and SMT-CBMC [11] if we consider the verification of embedded software. ESBMC is able to model check ANSI-C programs that involve tight interplay between non-linear arithmetic, bit operations, pointers and array manipulations. In addition, it was able to find undiscovered bugs in the NECLA, PowerStone, Siemens, SNU-RT, VERISEC and WCET benchmarks related to arithmetic overflow, buffer overflow, invalid pointers and pointer arithmetic.

SMT-CBMC still has limitations not only in the verification time (due to the lack of simplification based on high-level information), but also in the encodings of important ANSI-C constructs used in embedded software. CBMC is a SAT-based BMC tool for full ANSI-C, but it has limitations due to the fact that the size of the propositional formulae increases significantly in the presence of large data-paths and high-level information is lost when the VCs are converted into propositional logic (preventing potential optimizations to reduce the state space to be explored). Its prototype SMT-based back-end is still unstable and fails on a large fraction of our benchmarks.

Chapter 4

Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking

We describe and evaluate three approaches to model check multi-threaded software with shared variables and locks using bounded model checking based on Satisfiability Modulo Theories (SMT) and our modelling of the synchronization primitives of the Pthread library in order to achieve the second objective stated in Section 1.2. In the lazy approach, we generate all possible interleavings and call the SMT solver on each of them individually, until we either find a bug, or have systematically explored all interleavings. In the schedule recording approach, we encode all possible interleavings into one single formula and then exploit the high speed of the SMT solvers. In the underapproximation and widening approach, we reduce the state space by abstracting the number of interleavings from the proofs of unsatisfiability generated by the SMT solvers. In all three approaches, we bound the number of context switches allowed among threads in order to reduce the number of interleavings explored. We implemented these approaches in ESBMC, our SMT-based bounded model checker for ANSI-C programs. Our experiments show that ESBMC can analyze larger problems and substantially reduce the verification time compared to state-of-the-art techniques that use iterative context-bounding algorithms or counter-example guided abstraction refinement.

4.1 Introduction

Bounded model checking (BMC) has already been successfully applied to verify software and to discover subtle errors in real systems [24]. In an attempt to cope with growing

system complexity, Boolean Satisfiability (SAT) solvers are increasingly replaced by Satisfiability Modulo Theories (SMT) solvers to prove the validity of the generated verification conditions (VCs) [11, 53, 71]. Recently, there have also been attempts to extend BMC to the verification of multi-threaded software [73, 102, 103, 152]. The main challenge here is the state space explosion problem, as the number of possible interleavings grows exponentially with the number of threads and program statements. However, two important observations can help us. First, most concurrency bugs in real applications have been found to be shallow so that only a few context switches are required to expose them [150]. We can thus use a context-bounded analysis [112, 171] that limits the number of context switches it explores. Second, SAT and SMT solvers produce unsatisfiable cores that allow us to remove logic that is not relevant to a given property [129]. Grumberg et al. [84] showed that the unsatisfiable cores can also be used to control the number of allowed interleavings of the given set of processes. They proposed a SAT-based BMC method to model check a multi-process system using a series of under-approximated models. However, their method does not combine context-bounded analysis with symbolic algorithms, which limits its usefulness for verifying multi-threaded software. It has also not been applied in conjunction with the SMT solvers.

In the previous chapter, we extended the encodings from previous SMT-based BMC [11, 71] to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers. Here, we continue this work and develop and evaluate three related approaches for model checking multi-threaded ANSI-C software. In contrast to previous fully symbolic approaches (e.g., [73, 102, 103, 152, 84]), we combine symbolic model checking with explicit state space exploration. In particular, we explicitly explore the possible interleavings (up to the given context bound) while we treat each interleaving itself symbolically. This approach is similar to the recent ESST approach by Cimatti et al. [39], but we handle ANSI-C instead of SystemC, we use BMC instead of predicate abstraction, and place no restrictions on the scheduler. Our approaches all implicitly use the reachability tree (RT) derived from the system, but differ in the way they exploit it. In the *lazy* approach, we traverse the RT depth-first, and simply call the single-threaded BMC procedure on the interleaving whenever we reach an RT leaf node. We stop the RT traversal either when we find a bug, or have systematically explored all interleavings. In the *schedule recording* approach, we use the RT to encode all the possible execution paths into one single formula, which is then fed into the SMT solver. In a third approach, we extend the *under-approximation and widening* (UW) algorithm [84] with the purpose of addressing the verification of real-world C code using different background theories and SMT solvers.

This chapter makes two major novel contributions. First, we exploit SMT to improve BMC of multi-threaded software. We describe a comprehensive SMT-based BMC procedure to support the checking of multi-threaded C programs that use the synchronization

primitives of the POSIX Pthread Library [135]. Second, we describe and evaluate three related approaches to SMT-based BMC. This work also marks the first application of the UW algorithm in combination with context-bounded model checking to verify non-trivial multi-threaded C software. Experiments obtained with the extended ESBMC show that our approaches can analyze larger problems and substantially reduce the verification time compared to state-of-the-art techniques that use iterative context-bounding algorithms and others that implement counter-example guided abstraction refinement (CEGAR) techniques.

4.2 Preliminaries

In the widely adopted interleaving paradigm for multi-threaded programs, the notion of concurrency is represented by that of interleaving, i.e., the non-deterministic choice between activities of the simultaneously acting threads [40]. If only a single core is available, the actions of the different threads must obviously be interleaved on this core; however this concept also applies to multiple cores, as there are many different possible orderings between truly concurrent events [14]. An interleaving represents a possible execution of the program where all of the concurrent events are arranged in a linear order. Any change of the active thread in an interleaving is called a *context switch*.

The interleaving paradigm relies on a scheduler, which selects the concurrently executing threads according to a given strategy. This abstracts from the speed of the participating threads and thus models any possible realization by a single-core machine or by several cores with arbitrary speeds. However, in order to fully verify a multi-threaded program against a given specification, all possible interleavings must be considered. This results in a large state space that must be explored by a model checker.

4.2.1 Multi-threaded Goto Programs

We consider multi-threaded ANSI-C programs in asynchronous mode and assume that all threads in the program only communicate through shared global variables. ESBMC handles full ANSI-C, but for presentation, we use a minimal language similar to the internal *goto*-language of the CBMC model checker [42]. It is expressive enough to model multi-threaded programs. We summarize the language in Figure 4.1.

A multi-threaded *goto*-program is a (numbered) list of commands. Commands include assignments, non-deterministic assignments ($Var = *$), blocking statements ($assume$) to cut off subsequent executions paths, and assertion statements ($assert$) to indicate user-specified properties. All control structures are represented by explicit (conditional) jumps to a statement $l \in \{1, \ldots, n\}$. A thread t is a sublist of commands between *begin_thread* and *end_thread*. Threads are created via asynchronous procedure calls

$$
\begin{aligned}
Prop &::= Var \mid true \mid false \mid Prop \wedge Prop \mid \ldots \mid Exp = Exp \mid \ldots \\
Exp &::= Var \mid Const \mid Var[Exp] \mid Exp\ +\ Exp \mid \ldots \\
Cmd &::= skip \mid Var = Exp \mid Var = * \mid assume\ Prop \mid assert\ Prop \\
&\quad \mid goto\ l \mid if\ Prop\ goto\ l \mid begin_atomic \mid end_atomic \\
&\quad \mid begin_thread\ Id \mid end_thread \\
&\quad \mid Var\ =\ start_thread\ Id \mid join_thread\ Var \\
Prog &::= Cmd; \ldots; Cmd
\end{aligned}
$$

FIGURE 4.1: Multi-threaded Goto Program Language

(*start_thread*), which return an integer that can be used as thread identifier for synchronization (*join_thread*); hence, dynamic thread creation is allowed. Atomic statements (*atomic_begin* and *atomic_end*) indicate that a code segment cannot be preempted by another thread. Figure 4.2 shows an example of a multi-threaded C program and its representation in the multi-threaded goto-language. In this running example, we have three threads t_1, t_2 and *main*. Each thread contains one or more *effective statements*, i.e., statements that can influence the program state. In our minimal language, the only effective statements are assignments and assertions, since control-flow tests cannot influence the state. In the example in Figure 4.2(b), thread t_1 contains two effective statements (in lines 3 and 5), thread t_2 contains three (in lines 9, 10 and 12), while thread *main* contains one (in line 18).

4.2.2 Formal Model of Multi-threaded Software

The multi-threaded software to be analyzed is modelled as a tuple $M = (S, S_0, T, V)$ (cf. Definition 2.14), where:

- S is a finite set of states, with $S_0 \subset S$ the set of initial states;
- $T = t_0, t_1, ..., t_n$ is the set of threads, where n represents the total number of threads;
- $V = V_{global} \cup \bigcup V_j$ where V_{global} is the set of global variables and V_j is the set of local variables of t_j.

We assume that each variable ranges over a finite domain. A state $s \in S$ consists of the values of the global and local variables, including a local program counter for each thread. Each thread j is a tuple $t_j = (R^j, l^j)$, where:

- $R^j \subseteq S \times S$ is the transition relation of thread t_j;
- $l^j = \langle l_i^j \rangle$ is the sequence of thread locations l_i^j at time step i.

```
 1 #include <pthread.h>
 2 int x=0;
 3 void* t1(void* arg) {
 4   x++;
 5   if (x>1)
 6     x--;
 7   return NULL;
 8 }
 9 void* t2(void* arg) {
10   _Bool y;
11   x++;
12   y = (x>1);
13   if (y)
14     x--;
15   return NULL;
16 }
17 int main(void) {
18   pthread_t id1, id2;
19   pthread_create(&id1,NULL,t1,NULL);
20   pthread_create(&id2,NULL,t2,NULL);
21   pthread_join(&id1,NULL);
22   pthread_join(&id2,NULL);
23   assert(x==1)
24   return 0;
25 }
```

(a)

```
 1 int x = 0;
 2 begin_thread t1;
 3   x = x + 1;
 4   if !(x > 1) then goto L6;
 5     x = x - 1;
 6 L6: end_thread;
 7 begin_thread t2;
 8   _Bool y;
 9   x = x + 1;
10   y = x>1;
11   if !(y) then goto L13;
12     x = x - 1;
13 L13: end_thread;
14 id1 = start_thread t1;
15 id2 = start_thread t2;
16 join_thread id1;
17 join_thread id2;
18 assert(x==1);
19 return 0;
```

(b)

FIGURE 4.2: (a) A multi-threaded C program with an assertion violation. (b) The C program of (a) converted into multi-threaded goto form.

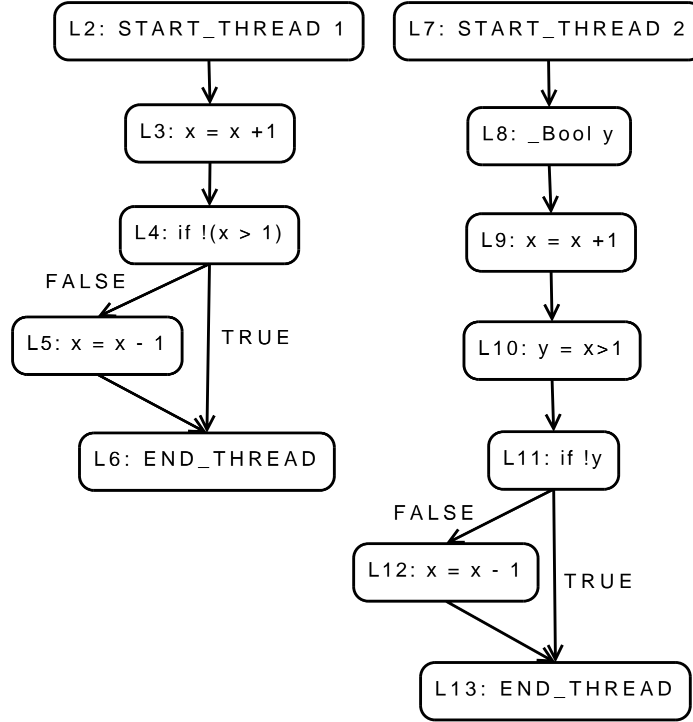

FIGURE 4.3: CFG of two threads of the goto program shown in Figure 4.2 (b).

The execution of the instructions of each thread t_j is modelled by means of transition relations and we use the notation $R_i^j(s, s')$ to denote that s' is a successor of s obtained by executing at time step i an instruction of thread t_j. We define $R_i(s, s') = \bigcup_j R_i^j(s, s')$ and $R(s, s') = \bigcup_i R_i(s, s')$. Finally, a particular program location, denoted by l_0^j is designated as the entry point of thread t_j.

4.2.3 Context-Bounded Encoding

As described in Section 2.3, our work considers multi-threaded programs in asynchronous mode and assumes that the threads in the program only communicate through shared (global) variables and synchronize to avoid the simultaneous access to shared variables. This means that at all times only one thread is running until a context switch occurs and another thread resumes its execution. Figure 4.4 shows an example of one possible concurrent execution of the two threads from Figure 4.3.

In our approach, we only consider *effective context switches*, i.e., context switches to effective statements. An ECS block then defines as a sequence of program statements that are executed with no intervening ECS. This definition is key to our context-bounded translation, because we only allow context switches before visible statements (i.e., before global variables and synchronization points). If the program statements are invisible, we group them into one ECS block thus reducing the number of possible concurrent executions.

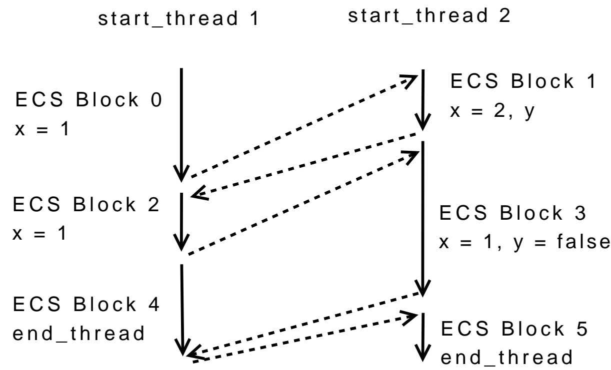

Figure 4.4: Concurrent execution of two threads.

In order to obtain a bounded multi-threaded C program, we bound the number of context switches between the ECS blocks up to C, as described in detail in the next sections. The technique is incomplete because there might still be a counterexample that requires more context switches than the specified context-bound C, but it is both sound and precise for context-bounded executions of multi-threaded programs. The technique of bounding the number of context switches was originally proposed by Qadeer and Rehof [150], but the authors apply this idea on Boolean programs using pushdown automata. Recently, a number of context-bounded translations for model checking Boolean [112, 171] and C programs [111, 152] have been proposed in the literature, but they neither use bounded model checking to generate the VCs nor SMT solvers to check the validity of the VCs.

4.3 Context-Bounded Model Checking of Multi-threaded Software

This section describes how to exploit SMT techniques to improve BMC of multi-threaded software. In particular, we exploit SMT solvers to prune the property and data dependent search space and to remove thread interleavings that are not relevant by analyzing proofs of unsatisfiability. We then propose three approaches to SMT-based BMC and show how the lazy, schedule recording, and UW approaches are encoded into BMC framework of multi-threaded software.

4.3.1 Exploring the Reachability Tree

In order to describe reachable states of a multi-threaded goto program, we use a reachability tree (RT) that is obtained by unfolding the set of running threads.

Definition 4.1. For a multi-threaded program with n active threads, each node in the RT is 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, where:

- A_i represents the currently active thread;
- C_i represents the context switch number;
- s_i represents the current state;
- l_i^j represents the current location of thread j;
- G_i^j represents the control flow guards accumulated in thread j along the path from l_0^j to l_i^j.

Since threads only communicate via global variables, we only need to consider context switches at visible instructions, i.e., synchronization points and statements containing global variables. As in Gupta et al. [73], we do not model context switches inside individual visible statements. This is safe as long as the statements only read or write a single global variable, but in general it is an under-approximation. However, we have not encountered any problems in the benchmarks we have used. Additionally, we do not model context switches between a visible control-flow test and the next visible statement, since the test cannot influence the state. However, note that we can simulate the effect of a context switch right after a visible test by hoisting the test out of the conditional, and assigning its result to a new auxiliary variable, as shown in thread t_2 in Figure 4.2(a). ESBMC can be configured to automatically insert such auxiliary variables. Finally, we also assume sequential consistency, as is common in model checking multi-threaded software [44, 102, 138, 152].

In order to expand the RT and explore all possible interleavings, we symbolically execute each instruction of the multi-threaded goto-program. This takes as input the program and the current RT node, and generates its children according to the set of rules described below. We assume that we expand an RT node ν at time step i and that the guard $G_i^{A_i}$ of the thread t^{A_i} is enabled in state s_i (i.e., that the corresponding formula is satisfiable), so that the thread can potentially execute the instruction I at location $l_i^{A_i}$.

R1 (ASSIGN): If I is an assignment $x = e$, then we symbolically execute I, which generates a new state s_{i+1}. We then add as child to ν a new node ν'

$$\nu' = (A_i, C_i, s_{i+1}, \langle l_{i+1}^j, G_i^j \rangle)_{i+1} \tag{4.1}$$

where the active thread remains unchanged. We increment the location of the active thread only (i.e., $l_{i+1}^{A_i} = l_i^{A_i} + 1$) and leave all other locations and all guards unchanged; however, note that the evaluation of the guards can change under the new state s_{i+1}, and hence threads may become enabled.

We have fully expanded ν if

- $l_i^{A_i}$ is within an atomic block; or
- I contains no global variable (since we allow context switches only at visible instructions); or
- we have reached the upper bound of context switches to be explored (i.e., $C_i = C$).

If ν is not yet fully expanded, we then also explore all context switches, up to the given context bound C. For each thread $j \neq A_i$ where G_i^j is enabled in s_{i+1}, we thus create a new child node

$$\nu_j' = (j, C_i + 1, s_{i+1}, \langle l_i^j, G_i^j \rangle)_{i+1} \tag{4.2}$$

In ν_j' we then continue the RT exploration with thread j executing in the state produced by the current thread A_i.

R2 (SKIP): If I is a *skip*-statement with target l, then we simply increment the location of the current thread and continue with it. However, we explore no context switches, i.e., we only add a single child node

$$\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, G_i^j \rangle)_{i+1} \tag{4.3}$$

where $l_{i+1}^j = l_i^j + 1$ only if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise.

R3 (unconditional GOTO): If I is an unconditional *goto*-statement with target l, then we simply set the location of the current thread and continue with it. However, we explore no context switches, i.e., we only add a single child node

$$\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, G_i^j \rangle)_{i+1} \tag{4.4}$$

where $l_{i+1}^j = l$ only if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise.

R4 (conditional GOTO): If I is a conditional *goto*-statement with test c and target l, then we create two child nodes ν' and ν'' for both possible outcomes of the test. For ν', we assume that c is *true* and proceed with the target instruction of the jump, similar to unconditional jumps. However, we also add c to the guards of all other threads, since it may contain global variables, and may thus enable or disable other transitions.[1] Hence, we construct

$$\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, c \wedge G_i^j \rangle)_{i+1} \tag{4.5}$$

[1] Note that any thread local variables in c are of course inaccessible to the other threads.

where $l_{i+1}^j = l$ if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise. For ν'', we add $\neg c$ to the guards and continue with the next instruction in the current thread, i.e.,

$$\nu'' = (A_i, C_i, s_i, \langle l_{i+1}^j, \neg c \wedge G_i^j \rangle)_{i+1} \tag{4.6}$$

where $l_{i+1}^j = l_i^j + 1$ if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise. We prune one of the nodes if the condition is determined in the current state (i.e., either evaluates to *true* or to *false*). Note that we are not exploring any possible context switches (even if I is visible), since the condition cannot change the global state.

R5 (ASSUME): If I is an *assume*-statement with argument c, then we proceed similar to the way described in R1. We continue with the unchanged state s_i but add c to all guards, as described in R4. If $c \wedge G_i^j$ evaluates to *false*, we prune the execution paths.

R6 (ASSERT): If I is an *assert*-statement with argument c, then we proceed similar to the way described in R1. We continue with the unchanged state s_i but add c to all guards, as described in R4. We also generate a verification condition to check the validity of c.

R7 (START_THREAD): If I is a *start_thread* instruction, we just add the indicated thread to the set of active threads, i.e., we add a node

$$\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, G_{i+1}^j \rangle_{j=1}^{n+1})_{i+1} \tag{4.7}$$

where l_{i+1}^{n+1} is the initial location of the indicated thread, and $G_{i+1}^{n+1} = G_i^{A_i}$, i.e., the thread starts with the guards of the currently active thread.

R8 (JOIN_THREAD): If I is a *join_thread* instruction with argument *Id*, then we add a child node

$$\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, G_i^j \rangle)_{i+1} \tag{4.8}$$

where $l_{i+1}^j = l_i^{A_i} + 1$ only if the joining thread *Id* has exited. We model this by an additional variable $exit_j$ that is set to *false* when *begin_thread Id* is called. When *end_thread* is reached, we set $exit_j$ to *true* to indicate that thread *Id* has exited.

The remaining instructions (*begin_atomic*, *end_atomic*, *begin_thread*, and *end_thread*) are just scoping constructs and do not contribute to the expansion of the RT. As example, we consider the C program with two threads and the corresponding goto-program, as shown in Figure 4.2(a) and (b). This example is modified slightly from Ghafari et al. [74], where it is used to check (by increasing the number of increments) the scalability of

different context-bounded analysis algorithms. Both threads increment a global variable x, and then, depending on the value of x, decrement it again. t_2 uses a local variable y to store the value of x and uses this in the test (cf. lines 12–13). This simulates a possible context switch between the evaluation of the guard and the execution of the next statement. Figure 4.3 shows the CFG representation of the two threads t_1 and t_2. Note that this example contains an assertion violation in line 23, where the invariant $x = 1$ does not hold under specific thread interleavings.

Figure 4.5 shows a fragment of the reachability tree for threads t_1 and t_2 (where t_0 represents the main thread). We build this by first executing the goto-program of Figure 4.2(b) sequentially, i.e., in the same order that the threads are created. In this case, we first execute the statements of t_1 (i.e., lines 3-5), followed by the statements of t_2 (i.e., lines 8-12). The initial node of the RT fragment is $\nu_0 = (t_0, 0, s_0, \langle (L_{16}, true), (L_2, true), (L_7, true) \rangle)$, i.e., the main thread t_0 is active at line 16, the program is before the first context switch, the state s_0 has $x = 0$ and y undefined, and both threads t_1 and t_2 have just been started, i.e., are at their initial location with guards *true*. To expand the RT, we check which threads are enabled from ν_0.[2] Since t_1 and t_2 are both enabled and since our approach always expands the enabled thread with the smallest index, we expand the transitions of t_1. The transition relation $R_1^1(s_0, s_1)$ of t_1 that represents the assignment $x = x + 1$ is defined as follows:

$$R_1^1(s_0, s_1) \Leftrightarrow l_1^1 = L3 \wedge x_1 = x_0 + 1 \wedge \forall v \in V \setminus \{x\} : v_1 = v_0$$

The first term corresponds to the unconditional edge from line 2 to 3 (see Figure 4.3). The second term defines the new value of the shared variable x. The third term ensures that the values of V, but not x, do not change in the transition from s_0 to s_1. To create node ν_1, we apply rule R1, which gives us $\nu_1 = (t_1, 1, s_1, \langle (L_{16}, true), (L_3, true), (L_7, true) \rangle)$. We then check again which threads are enabled and expand t_1 as the enabled thread with the smallest index. The transition relation that represents the branch at program location L4 is defined by a case-split on the value of x in state s_1.

$$R_2^1(s_1, s_2) \Leftrightarrow l_2^1 = \begin{cases} L6 : \neg(x_1 > 1), \\ L5 : otherwise \end{cases}$$
$$\wedge \forall v \in V : v_2 = v_1$$

The transition does not affect the global state (as the condition $\neg(x_1 > 1)$ holds), so we only increment the program location but do not create a new node in the RT (described in rule R4). Therefore, to expand the next node from ν_1, we check again which threads are enabled and since t_1 has executed all its statements, we then expand the first instruction of thread t_2. The transition relation $R_3^2(s_2, s_3)$ of t_2 is similar to $R_1^1(s_0, s_1)$. We thus apply

[2] We ignore interleavings with t_0 to simplify the presentation.

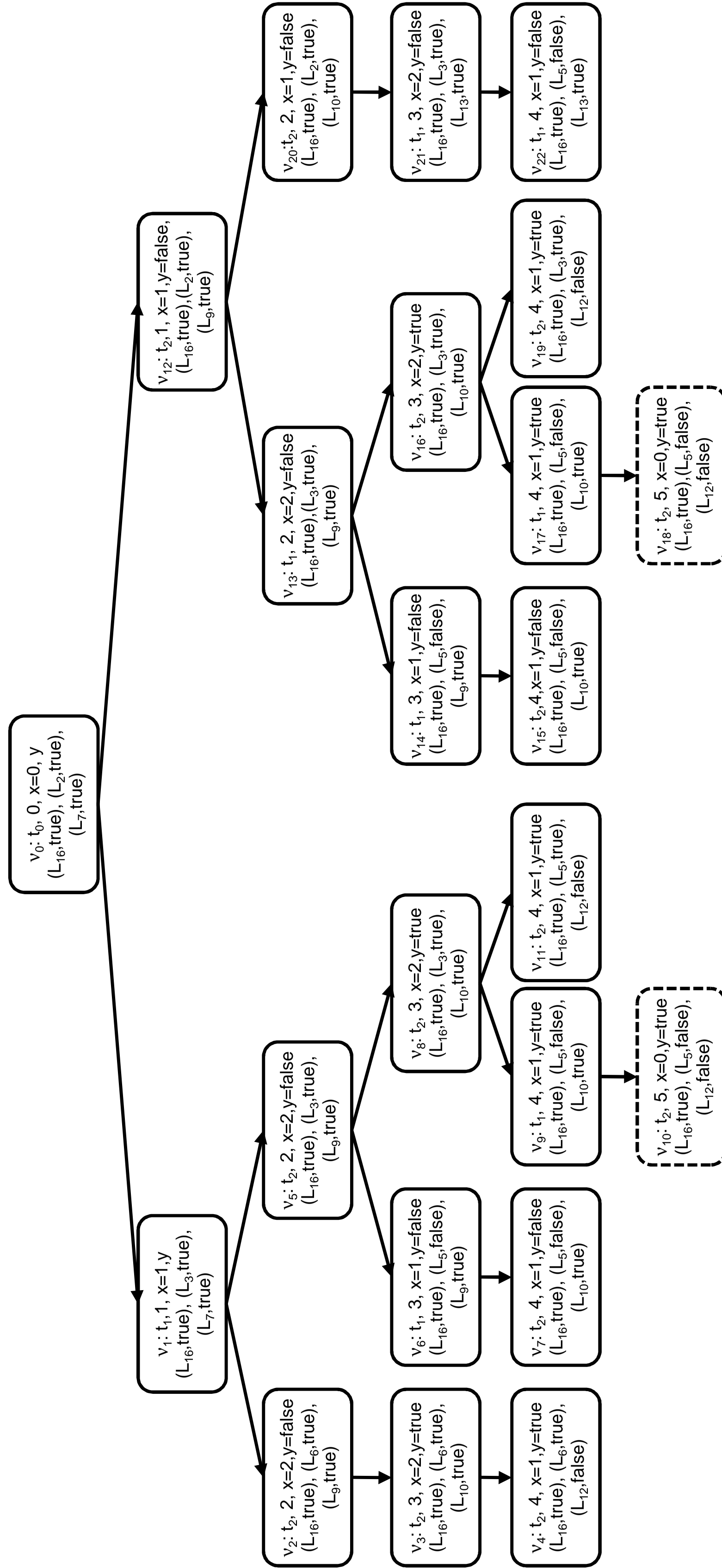

FIGURE 4.5: Fragment of the reachability tree of the multi-threaded goto-program of Figure 4.2(b). Nodes with dashed line represent program locations that violate the assertion statement in line 18 of Figure 4.2(b).

rules R1 and R2 to derive $\nu_2 = (t_2, 2, s_2, \langle (L_{16}, true), (L_6, true), (L_9, true)\rangle)$. ν_3 and ν_4 are derived in the same way. After creating ν_4, both t_1 and t_2 do not have enabled transitions and we backtrack to explore pending transitions from previous nodes; in this case, we have already explored ν_3 and ν_2 and continue the RT exploration at ν_1.

4.3.2 Lazy Approach

The idea of the lazy approach to verify multi-threaded software is to traverse the RT depth-first, and to call the single-threaded BMC procedure on each interleaving whenever we reach an RT leaf node. We stop the RT traversal either when we find a bug, or have systematically explored all interleavings. This approach seems obvious, but to the best of our knowledge, it has not been formalized nor evaluated in the literature. Figure 4.6 details how the lazy approach works. Formally, given an RT $\Upsilon = \{\nu_1, \ldots, \nu_N\}$ that represents the program unfolding for a context bound C and a bound k, and a property ϕ, we derive a VC ψ_k^π for a given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. As always in our work, the VC ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. The model checking problem associated with SMT-based BMC of a given π is then formulated by constructing the logical formula [11, 71]:

$$\psi_k^\pi = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{constraints} \wedge \overbrace{\neg\phi_k}^{property} \tag{4.9}$$

Here, ϕ_k represents a safety property ϕ in step k, I is the function for the set of initial states of M and $R_i(s_i, s_{i+1})$ is the function representing the transition relation of M at time steps i and $i+1$, as described by the states in the nodes of π. If ψ_k^π is satisfiable, then ϕ is violated and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counterexample. A counterexample for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, and $R(s_i, s_{i+1})$ for $0 \leq i < k$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in length k along π.

On the face of it, the lazy approach seems to be naive: despite the context-bounding, the RT and thus the number of interleavings can grow very quickly, and we need to invoke the SMT solver several times to check the satisfiability of formula (4.9), which might slow down the verification process. However, there are several observations that make this approach worthwhile. First, if the program contains any errors at all, they will often be exhibited in a substantial fraction of the interleavings (cf. Qadeer and Rehof [150] and our evaluation in Section 4.6 for experience on benchmarks and applications), so that in practice we only need to explore a small part of the search space until we find the first

Step 1: Initialize the stack with the initial node ν_0 and the initial path $\pi_0 = \langle \nu_0 \rangle$.

Step 2: If the stack is empty, terminate with "no error".

Step 3: Pop the current node ν and current path π off the stack and compute the set ν' of successors of ν using rules R1-R8.

Step 4: If ν' is empty, derive the VC ψ_k^π for π using formula (4.9), and call the SMT solver on it. If ψ_k^π is satisfiable, terminate with "error"; otherwise, goto step 2.

Step 5: If ν' is not empty, then for each node $\nu \in \nu'$, add ν to π, and push node and extended path on the stack. Goto step 3.

FIGURE 4.6: Algorithm of the lazy approach.

error. In our running example, the invariant $x = 1$ does not hold for the two nodes ν_{10} and ν_{18} and if we traverse the RT depth-first and left-to-right, the error already shows up in the third interleaving. Second, we do not need to actually build the entire RT; instead, we only keep in memory nodes on computation paths that are still unexplored and expand them one path at a time. We then construct the VC for the chosen computation path and feed it into the SMT solver to check for satisfiability. Third, and most important, we can leverage the optimizations from the ESBMC front-end (e.g., constant folding and constant propagation as described in Chapter 5) to exploit which transitions are enabled in a given state to drive the exploration of the interleavings and to reduce both the number of interleavings to be explored and the size of the formulas sent to the SMT solver. For example, if we continue to explore thread t_1 from node ν_1, the front-end exploits the fact that $x = 1$ to infer that the guard in line 4 holds. t_1 thus continues in line 6, and terminates, so that the exploration continues with a context switch to thread t_2, as shown in node ν_2. Note that our current implementation does not check the satisfiability of the accumulated guards, and simply assumes that all running threads are enabled, unless they have explicitly been blocked or their guards evaluate to *false*. Implementing this could further reduce the size of the RT to be explored.

In summary, the lazy approach guides the symbolic execution between the threads and systematically explores all the possible interleavings in a lazy way. This approach can find bugs fast and the VCs are relatively small, since they correspond to a single interleaving only, but as the front-end invokes the SMT solver, once for each possible computation path, it can suffer performance degradation, in particular for correct programs where we have to explore all possible interleavings.

4.3.3 Schedule Recording Approach

State-of-the-art SMT solvers are built on top of efficient SAT solvers to speed up the performance on large problems by exploiting the support for conflict clauses and non-chronological backtracking [163]. In the schedule recording approach we leverage this

and avoid invoking the SMT solver repeatedly. We thus build the RT as before to systematically explore the interleavings, but we now add schedule guards [103] to record in which order the scheduler has executed the program. Figure 4.7 shows how schedule guards are added to the program during the exploration of the left-hand side of the RT in Figure 4.5. We then encode all interleavings into a *single* large formula, which is finally passed to the SMT solver.

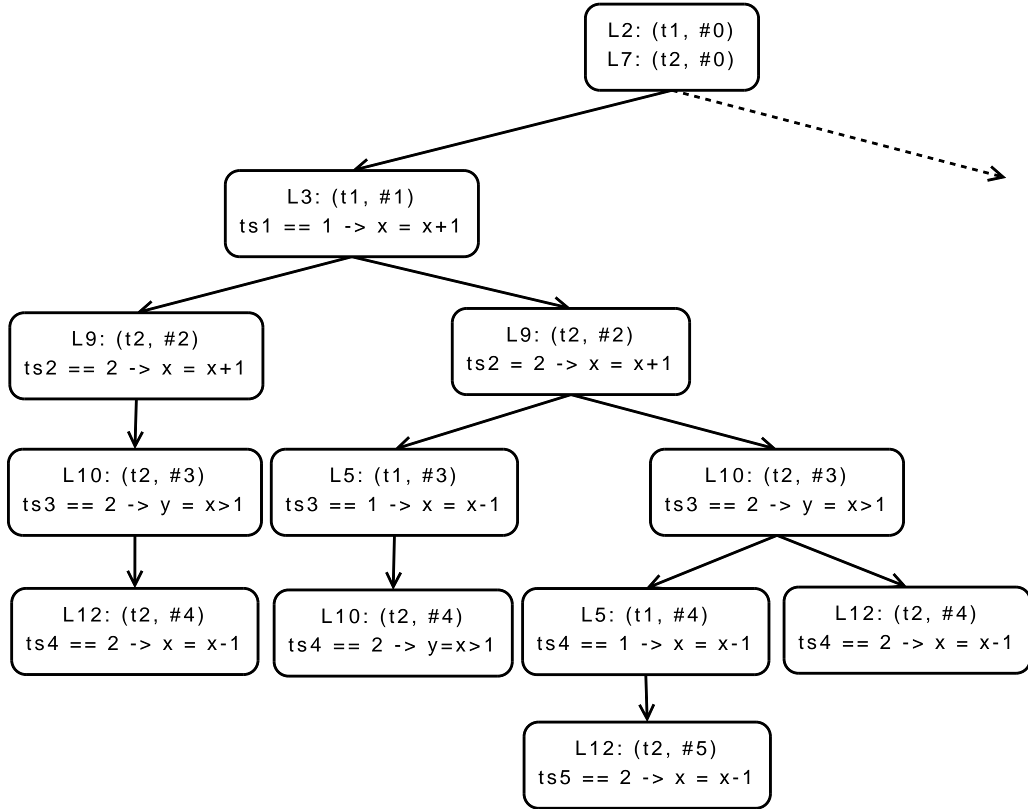

FIGURE 4.7: Schedule recording applied to the left-hand side of the RT in Figure 4.5.

Since control-flow tests cannot influence the state, we only need to add guards to *effective statements*, i.e., assignments and assertions (as described in Section 4.2.3). Each effective program statement is then prefixed by a schedule guard $ts_i = j$ where ts_i is the thread selection variable for the i-th ECS and j is the thread identifier. Its intuitive interpretation is that the statement can only be executed if thread j is scheduled to run after the i-th ECS. For example, the schedule guard $ts_1 = 1$ at L_3 encodes that the assignment $x = x + 1$ can only be executed if t_1 runs after the first ECS.

The schedule guards are added when program statements are executed symbolically and become part of the produced verification conditions. They can be derived from the RT nodes, i.e., for node ν_i we construct the guard $ts_{C_i} = A_i$. The thread selection variables are free variables that the SMT solver will instantiate with concrete values. The instantiation of all thread selection variables corresponds to the choice of a specific interleaving. In our running example, if the SMT solver chooses $ts_1 = 1$, $ts_2 = 2$, $ts_3 = 2$, and $ts_4 = 2$, then the model checker simulates the effect of executing the

program statements at L_3, L_9, L_{10}, and L_{12} (in that order). Note that the ordering of statements within a thread is of course still ensured by the program order semantics, so that the program statement at L_{10} will not be executed before the program statement at L_9 (i.e., we ensure sequential consistency [44, 102, 138, 152]). We further define a schedule SCH to determine which interleavings should be considered and encode the guards in (4.10) as:

$$\psi_k = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{constraints} \wedge \overbrace{\neg\phi_k}^{property}$$
$$\wedge \overbrace{SCH(s_0) \wedge \ldots \wedge SCH(s_k)}^{scheduler} \tag{4.10}$$

Here $SCH(s_i)$ represents a constraint on the schedule guard of state s_i. If we do not impose any schedule constraints, then we formulate $\bigwedge_{i=0}^{k} SCH(s_i) = true$ and all possible interleavings are considered. However, if we want to apply aggressive reductions (for example by exploiting the proofs of unsatisfiability as described in the next subsection), we can add constraints to SCH to force the removal of interleavings that do not contribute to checking a given property. Although we can bound the number of preemptions and exploit which transitions are enabled in a given state when we build formula (4.10), the number of threads and context switches can still grow very large quickly, and easily lead to formulae that overwhelm the solver.

4.3.4 UW Approach

The core idea of the under-approximation and widening (UW) approach is to check models with an increasing set of allowed interleavings [84]. We start from an underapproximation describing a single interleaving and widen the model by adding more interleavings incrementally based on the proof objects generated from an SMT solver [57]. We thus exploit the SMT solvers to remove possible undesired models of the program in order to satisfy a given property. This is possible because the SMT solvers can conclude that a given model is unsatisfiable without even using all of its constraints (since some of them might be redundant).

We define ψ' as an underapproximated model of ψ, i.e., $\psi' = \psi \wedge SCH(s_0) \wedge \ldots \wedge SCH(s_k)$, where we introduce constraints on the schedule guards. We can see that if ψ is unsatisfiable, then ψ' is also unsatisfiable; however, it is possible that ψ is satisfiable while ψ' is not, due to the constraints on the schedule. Thus, ψ' can be thought of as an underapproximation of ψ and each satisfying assignment of ψ' is also a satisfying assignment to ψ. The main steps of the UW algorithm are shown in Figure 4.8.

The additional literals cl_{ij} introduce constraints on the schedule guards (e.g., $cl_{ij} \rightarrow ts_i = j$), which allow us to guide the widening process according to the variables that

Step 1: Add control literals cl_{ij} (where i is the ECS number and j is the thread identifier) to the VC ψ_k.

Step 2: Add negated control literals $\neg cl_{ij}$ to the schedule *SCH*, except those enabling the first interleaving.

Step 3: Check satisfiability of ψ_k; if ψ_k is satisfiable, then terminate with "error".

Step 4: Check whether the proof objects generated by the SMT solver contains any control literals; if not terminate with "no error".

Step 5: Remove literals that are contained in the proof objects from the schedule *SCH* and go to step 3.

FIGURE 4.8: Algorithm of the UW approach.

participate in the proof of unsatisfiability produced by the SMT solver. This means that the schedule is now updated based on the information extracted from the proof, which aims to remove interleavings that are not relevant for checking a given property [84, 129]. Note that the way that we encode the underapproximation differs from Grumberg et al. [84]. Grumberg et al. encode an underapproximation using $m \times n$ control literals, where m is the number of control points that guard each program statement and n is the number of threads. In our encoding, we use $e \times n$ control literals, where e is the number of ECS (with $e \leq m$) and n is the number of threads. If we were to include a control literal for each statement as in [84], then our solution might not scale in practice to large multi-threaded software systems.

4.3.5 Pruning the RT with Partial Order Reduction

In the modelling of multi-threaded software, we consider that any of the threads $j \in T$ is able to make a transition and then we have to compute all states for which a thread j exists. The problem is that the number of states to be explored can grow dramatically with the number of program statements and threads. The purpose of the Partial-Order Reduction (POR) technique [40, 75, 146] is to reduce the number of states that have to be explored. This is done in a way that if the property holds on the reduced model, it also holds on the original model.

In our SMT-based BMC framework, as threads communicate only through global variables, we apply partial order reduction techniques at two levels in our algorithm. At the first level, we apply the visible instruction analysis POR (VI-POR) [146], which removes the interleavings of instructions that do not affect the global variables (i.e., we remove transitions which are independent from transitions made by any other thread). As we mentioned in Section 4.3.1, an instruction is *visible* only if it accesses a global variable, and it is invisible otherwise. VI-POR is "hard-wired" into our approach, due to the way we build the ECS blocks.

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

(a)

```
x = 0;
y = 0;
begin_thread t1;
  x = x + 1;
end_thread;
begin_thread t2;
  x = x + 1;
end_thread;
begin_thread t3;
  y = y + 1;
end_thread;
id1 = start_thread t1;
id2 = start_thread t2;
id3 = start_thread t3;
return 0;
```

(b)

FIGURE 4.9: (a) A simple multi-threaded C program. (b) The C program of (a) converted into goto form.

At the second level, we apply the read-write analysis POR (RW-POR) [49] in which two (or more) independent interleavings can be safely merged into one. As example, we consider a simple goto program with three threads and two global variables (x and y) as shown in Figure 4.9. Note that the global variable x in Figure 4.9 is shared between threads t_1 and t_2 only while the global variable y is only accessed by thread t_3. Figure 4.10 shows the reachability tree for threads t_1, t_2, and t_3. We build this RT by applying the rules R1-R8 as described in Section 4.3.1.

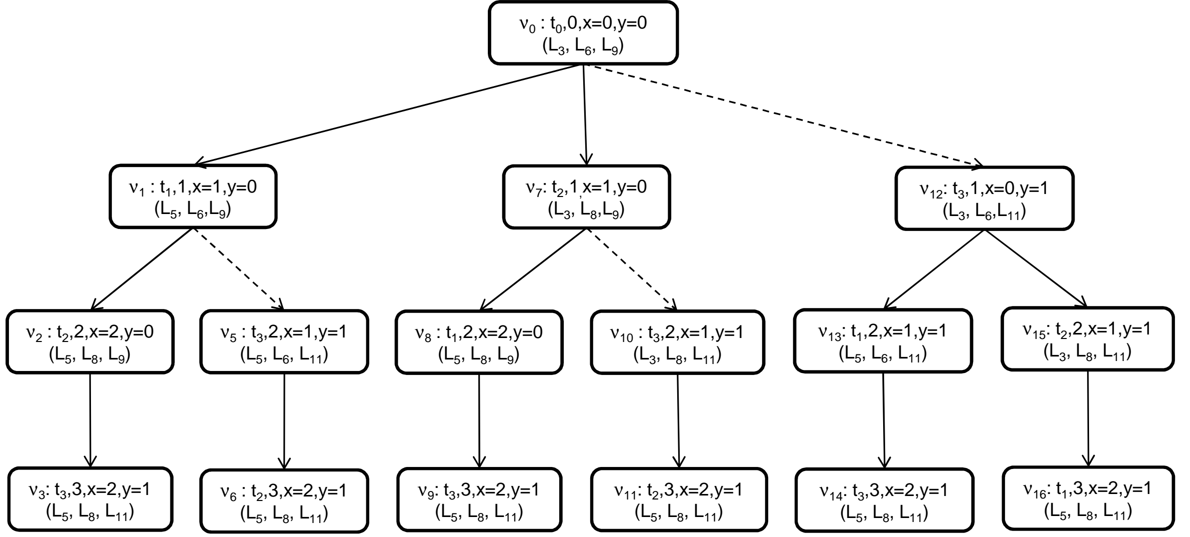

FIGURE 4.10: The reachability tree for threads t_1, t_2, and t_3 of the multi-threaded goto-program of Figure 4.9(b). Edges with dashed line represent transitions that can be eliminated by RW-POR.

In order to implement the RW-POR technique, we compute the sets of variables written (WR_j) and read (RD_j) by each of the threads. In particular, if

$$WR_j \cap (\bigcup_{k \neq j} RD_k \cup WR_k) = \emptyset \tag{4.11}$$

and

$$RD_j \cap \bigcup_{k \neq j} WR_k = \emptyset \tag{4.12}$$

i.e., if the intersection between the set of visible variables that are written and read by thread j and all other threads is empty, then we only explore the successors generated by executing j while all other transitions can be safely ignored.

For instance, in Figure 4.10 we get the node ν_1 from the initial node ν_0 after executing the program statement $x = x + 1$ of thread t_1. We can see that from node ν_1, we still have statements from threads t_2 and t_3 to execute. However, since thread t_3 does not share any global variable with thread t_1, then we can safely ignore the transition from node ν_1 to ν_5 (and consequently from node ν_5 to node ν_6). This reduction is safe because the different order of execution between the statements of threads t_2 to t_3 (or vice-versa) from node ν_1 always results in the same state. Hence, the RW-POR technique exploits the commutativity of concurrent transitions that result in the same state when they are executed in different orders. In our example, the transitions that can be safely eliminated by applying the RW-POR technique are indicated by edges with dashed line, as shown in Figure 4.10. Figure 4.11 shows the RT of Figure 4.10 after applying the RW-POR

technique.

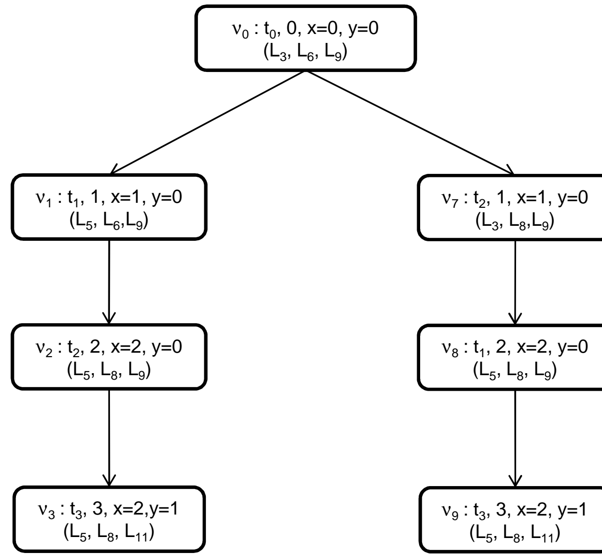

FIGURE 4.11: The reachability tree for threads t_1, t_2, and t_3 after applying the RW-POR technique.

In summary, there are six possible combinations of visible instructions of different threads, as shown in Table 4.1. There are three particular situations to consider when we build the reachability tree, as follows:

1. Two read operations from the same global variable, but from different threads will not modify the state, so they will always generate equivalent interleavings.
2. Two program statements accessing different variables are independent w.r.t. their execution states, thus these two program statements always generate equivalent interleavings with both execution orders.
3. Two program statements accessing the same global variable in such a way that at least one of them is a write access (i.e., with read-write and write-write relations) will generate non-equivalent interleavings.

In all three cases, the *read-write* relation actually causes *read-write* races and the *write-write* relation causes the *write-write* races. Consequently, only two types of relations will generate non-equivalent interleavings, while all other four types of relations generate equivalent interleavings. Those redundant interleavings are simply removed in our approach.

PORs work best in conjunction with an alias analysis. Our algorithms are able to remove redundant interleavings originating from pointer aliasing by dereferencing the actual thread parameters before building the reachability tree. This means that when a given thread is created with an argument (e.g., a pointer to a *void* type) and this

Access Relations to	Read-Read	Read-Write	Write-Write
Same variable	Equivalent	Non-equivalent	Non-equivalent
Different variables	Equivalent	Equivalent	Equivalent

TABLE 4.1: Read-write analysis of interleaving equivalence between visible instructions.

argument is used by the thread, we first get the object that the pointer points to before we apply the POR algorithms to build the reachability tree.

4.4 Verifying Race Conditions and Atomicity Violations

Concurrency bugs are tricky to reproduce and debug because they usually occur under specific thread interleavings. When verifying multi-threaded programs, it is important to detect data race conditions and atomicity violations, which are consistently ranked as the most common and difficult source of concurrency faults [61, 117]. This section presents our instrumentation to check for data race and atomicity violations in the multi-threaded goto programs.

4.4.1 Detecting Data Races

In a multi-threaded program with shared variable communication between the threads, data race conditions occur when multiple threads perform unsynchronized accesses to the shared variable [67, 139, 155, 158, 161]. In particular, a data race occurs when two (or more) threads access a shared variable at the same time and at least one of them is a write access. In a multi-threaded program, data races are often manifestations of bugs, because they may cause the program to behave in ways that are not expected by the developers.

There are two situations where data races may occur when two threads have access to the same shared variable simultaneously. In *read-write races*, one of the operations is read and the other one is write. Here, data race occurs because the value is changed by the write operation at the same time when the value is read by the read operation. In *write-write races*, both operations are writing (different) values to the same variable. Here, data race occurs because the first value written is overwritten by the other write operation.

We can identify both types of data races by breaking visible statements into two stages. In the first stage, we copy the value of the global variable into a local temporary variable and allow context switch. In the second stage, we check if the current value of the variable is the same as the copied value and if so we perform the assignment; otherwise we have detected an error (i.e., the assertion in the atomic section is violated). Figure 4.12

and 4.13 show our modelling of data race conditions for read and write operations (where g means a global variable while l means a local variable), respectively.

```
tmp = g;
atomic {
  assert(tmp==g);
  l = g;
}
```

FIGURE 4.12: Modelling data race conditions for read operations (l = g).

```
tmp = g;
atomic {
  assert(tmp==g);
  g = l;
}
```

FIGURE 4.13: Modelling data race conditions for write operations (g = l).

Visible statements that contain structs, arrays, and pointers are treated similarly. For example, if there is an assignment $l = *p$ where a pointer p points to a global variable g, we should first assign $*p$ to tmp, allow a context switch and then check whether $tmp = *p$. If the assertion fails, we have detected a data race condition; otherwise we simply perform the assignment $l = *p$.

4.4.2 Checking Atomicity

Atomicity, which is also referred as *serializability*, of program statements is satisfied only if the resulting state of data in a concurrent execution is the same as that of a serialized execution (i.e., if a thread interleaving executes a program statement without other threads interleaved in between) [144, 177]. An atomic block is a sequence of statements whose execution is not intervened by other threads. A program statement that contains more than one (global) variable access is not always executed as an atomic block. The assembly code generated by the compiler might break the statement into instructions so that a context switch may occur between these instructions. For example, the program statement $result = x + y$ can be broken by the compiler into four different (assembly) instructions as follows:

$$\begin{aligned} reg_1 &= x && (MOV\ reg_1\ ,\ \#x) \\ reg_2 &= y && (MOV\ reg_2\ ,\ \#y) \\ reg_2 &= reg_1 + reg_2 && (ADD\ reg_2\ ,\ \#reg_1) \\ result &= reg_2 && (MOV\ result\ ,\ \#reg_2) \end{aligned}$$

The instruction $MOV\ reg_1,\ \#x$ moves the content of x to reg_1 while $ADD\ reg_2,\ \#reg_1$

sums the content of registers reg_1 and reg_2 and stores the result in reg_2. Note that as a context switch may occur between these instructions, we have to consider this behaviour by modelling visible statements that contain more than one global variable just as a compiler does. However, from the verification point of view, statements that read or write a single global variable are not affected by context switches.

We thus implement a procedure to break statements with multiple global variables so that our approach can produce sound results. In particular, we break a visible statement into two atomic blocks. The first block contains temporary variables to store each variable of the right-hand side of the visible statement. The second block checks whether the temporary variables are equal to the variables of the right-hand side of the visible statement and if so we perform the assignment; otherwise we have detected an atomicity violation. Note that a context switch may occur between the first and second atomic block (but only there).

As example, consider the program statement $result = x + y$, where x and y are global variables. Figure 4.14 shows how we model atomicity violation at statement $result = x + y$. We first assign x and y to the temporary variables tmp_1 and tmp_2, respectively. After that, we allow a context switch and we then check whether $tmp_1 == x$ and $tmp_2 == y$.

```
atomic {
  tmp1 = x;
  tmp2 = y;
}
atomic {
  assert(tmp1==x && tmp2==y);
  result = x + y;
}
```

FIGURE 4.14: Modelling atomicity violation at visible statements.

Note further that statements that involve conditionals and loops are treated similarly, i.e., if the condition accesses more than one global variable, we hoist the statement out of the conditional, and then break it into two atomic blocks.

4.5 Modelling Synchronization Primitives in Pthread

This section presents our modelling of the synchronization primitives of the Pthread library [135]. We assume that the library function implementations are correct and focus our effort on verifying only the client programs that use them. We thus provide an instrumented model of the Pthread functions and use this to model check the client code. We show, in our experiments, that our modelling is able to detect incorrect use of the functions and is also able to detect blocking operations that can lead to local and global deadlocks.

4.5.1 Modelling Mutex Locking Operations

The Pthread library supports two functions to implement mutual exclusion between threads called *pthread_mutex_lock* and *pthread_mutex_unlock* [143]. Both functions take as argument a data structure called *mutex* that has two states, “locked” and “unlocked”. The function *pthread_mutex_lock* locks the mutex if it is unlocked; otherwise it blocks the current thread until the mutex is unlocked and can successfully be locked again. The function *pthread_mutex_unlock* simply unlocks a locked mutex. Computation paths are blocked on a mutex when a thread tries to lock a mutex that has already been locked by another thread. As an example, consider the threads t_A and t_B, which both lock and unlock the same mutex m, as shown in Figure 4.15. The paths $A_0; A_1; B_0; B_1$ and $B_0; B_1; A_0; A_1$ are non-blocking or *wait-free* while the other two paths are blocked.

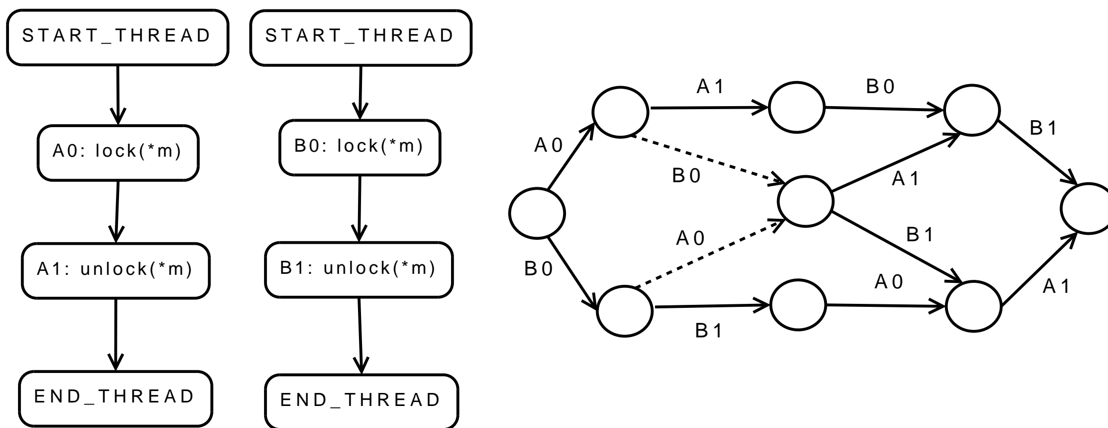

FIGURE 4.15: Computation paths blocking on a mutex.

A strategy to model mutex operations based on the notion of wait-free paths was proposed by [151, 152]. Instead of blocking the computation paths starting with $A_0; B_0$ and $B_0; A_0$, they are simply ignored by modelling the function *pthread_mutex_lock*(m) as

$$atomic\,\{assume(*m == 0); *m = 1\}$$

where the statement $assume(*m == 0)$ cuts off subsequent paths if the mutex is already locked. *pthread_mutex_unlock*(m) is then modelled as

$$atomic\,\{assert(*m == 1); *m = 0\}$$

which simply checks if the mutex is already locked. If so, the lock is released; otherwise, a thread tries to unlock a mutex that has not been locked previously, and we have detected an error.

This is sufficient to find bugs related to data races and lock acquisition ordering, but not to detect local and global deadlocks [151, 152]. We thus model *pthread_mutex_lock*

in such a way that we can detect global and local deadlock caused by the wrong use of the mutexes; *pthread_mutex_unlock* remains unchanged. For this, we first need to look in more detail at the different possible states that our model allows for a thread:

1. *Join state:* the thread is waiting for another thread to terminate.
2. *Lock state:* the thread is waiting for a mutex to be unlocked.
3. *Wait state:* the thread is waiting for a signal or broadcast to wake up.
4. *Exit state:* the thread has already exited.
5. *Free state:* the thread is not in any of the above four states and is free to execute its instructions.

The wait state is introduced to handle synchronization via signal/wait (see Section 4.5.2). A thread is blocked if it is in one of the *join*, *lock* or *wait* states, and is supposed to be running if it is not in *exit* state. Global deadlock occurs when all threads wait for a mutex and a local deadlock occurs when some of the threads form a waiting cycle. In both cases, we can detect the deadlock if there is no running thread in the free state, i.e., the number of blocked threads is equal to the number of running threads.

Figure 4.16 presents our modelling to detect global and local deadlock with mutexes, which maintains counts on both blocked and running threads with global variables.

```
1  int pthread_mutex_lock(pthread_mutex_t *m) {
2  extern uint trds_in_run, c_lock=0;
3    atomic {
4      unlocked = (mutex_lock_field(*m)==0);
5      if (unlocked) mutex_lock_field(*m) = 1;
6      else c_lock = c_lock + 1;
7    }
8    atomic {
9      if (mutex_lock_field(*m)==0)
10       c_lock = c_lock - 1;
11     if (!unlocked) {
12       deadlock_mutex = (c_lock<trds_in_run);
13       assert(deadlock_mutex);
14       assume(!deadlock_mutex);
15     }
16   }
17   return 0;
18 }
```

FIGURE 4.16: Modelling mutex lock operation.

mutex_lock_field retrieves the state of the mutex. We also use the variable *c_lock* to count the number of threads that are in the lock state due to mutex m, and *trds_in_run* to count the number of threads that are currently running. Initially, the mutex is unlocked

and we only lock it after the first call to *pthread_mutex_lock*. In subsequent calls, we increase the value of the variable *c_lock*, allow context switches, check if the mutex m was unlocked, and then assert $c_lock < trds_in_run$. If the assertion fails, a deadlock was detected: a thread is blocked by a lock operation on a mutex and the required mutex never gets unlocked by the thread that owns it, either because the locking thread has exited or because it has been blocked by another operation. If the assertion holds, we then eliminate this execution as described above.

As example, Figure 4.17 shows a code fragment extracted (and slightly modified) from the INSPECT suite [182], which aims to capture a concurrent scenario typically used in database systems, as described in [183]. For the sake of simplicity, we show the code for threads t_1 and t_2 only, which support two distinct classes of operations A and B (see lines 9 and 20) on a shared database. The intuitive interpretation of these operations is that the threads can run concurrently only if they belong to the same operation class. Here, the global variables A and B count the number of threads that are performing operations A and B respectively. The mutex l is used for the mutual exclusion between threads of distinct classes, while the mutex m is used for the mutual exclusion between threads of the same class.

The code shown in Figure 4.17 tries to implement the concurrent scenario described above, but it contains a subtle error (i.e., a local deadlock) that is exposed only under specific thread interleavings. Since we have four threads and two locked (i.e., $trds_in_run = 4$ $c_lock = 2$), our approach does not detect the local deadlock immediately. A local deadlock is detected only when the exploration of the running threads (that are not in deadlock) terminates and the invariant $c_lock < trds_in_run$ becomes *false*.

One possible thread interleaving to expose this error is to execute the program statements of threads t_1 and t_2 in the following order: $t_{1,5}$, $t_{1,6}$, $t_{1,7}$, $t_{2,16}$, $t_{2,17}$ and $t_{2,18}$ (the term $t_{j,i}$ denotes that j is the thread identifier and i is the program statement). The full counterexample produced by our model checker is shown in Appendix D.

4.5.2 Modelling Conditional Waiting

We model the functions *pthread_cond_wait*, *pthread_cond_signal*, and *pthread_cond_broadcast* from the Pthread library that implement conditional waiting [143]. All functions take as argument a condition variable c that has also two states, "locked" and "unlocked"; *pthread_cond_wait* also takes a mutex argument. Our modelling of the conditional waiting operation again employs the notion of wait-free execution paths. *pthread_cond_wait* is used to block the thread on a condition variable; the blocked thread is woken up only if another thread calls signal or broadcast. If several threads are blocked on a condition variable, then *pthread_cond_signal* non-deterministically unblocks at least one of them

```
1  #include <pthread.h>
2  pthread_mutex_t  m, l;
3  int A = 0, B = 0;
4  void *t1(void *arg) {
5    pthread_mutex_lock(&m);
6    A++;
7    if (A == 1) pthread_mutex_lock(&l);
8    pthread_mutex_unlock(&m);
9    //perform class A operation
10   pthread_mutex_lock(&m);
11   A--;
12   if (A == 0) pthread_mutex_unlock(&l);
13   pthread_mutex_unlock(&m);
14 }
15 void *t2(void *arg) {
16   pthread_mutex_lock(&m);
17   B++;
18   if (B == 1) pthread_mutex_lock(&l);
19   pthread_mutex_unlock(&m);
20   //perform class B operation
21   pthread_mutex_lock(&m);
22   B--;
23   if (B == 0) pthread_mutex_unlock(&l);
24   pthread_mutex_unlock(&m);
25 }
26 ...
27 int main(void) {
28   ...
29   pthread_create(&id1, NULL, t1, NULL);
30   pthread_create(&id2, NULL, t2, NULL);
31   pthread_create(&id3, NULL, t3, NULL);
32   pthread_create(&id4, NULL, t4, NULL);
33   ...
34 }
```

FIGURE 4.17: An example of local deadlock with mutex on a database application.

while *pthread_cond_broadcast* unblocks all threads blocked on the specified condition variable.

Figure 4.18 shows our modelling for the wait-operation. We use the variable *c_wait* to count the number of threads that are in the wait state due to condition c. Whenever a thread calls *pthread_cond_wait*, we atomically lock the condition variable c, assert that the mutex m is currently locked, release the mutex (so that other threads that access it can make progress), and then increment the number of threads in wait state (i.e., threads that are waiting for a signal or broadcast to wake up). We then allow context switches before we check whether the number of threads in wait state is less than the total number of the threads that are currently running with the assertion $c_wait < trds_in_run$. If the assertion holds or the variable c is locked, we simply eliminate this execution as described above.

```
 1 int pthread_cond_wait(pthread_cond_t *c,
 2                          pthread_mutex_t *m) {
 3 extern uint trds_in_run, c_wait=0;
 4   atomic {
 5     cond_lock_field(*c) = 1
 6     assert(mutex_lock_field(*m))
 7     mutex_lock_field(*m) = 0
 8     c_wait = c_wait + 1
 9   }
10   atomic {
11     deadlock_wait = (c_wait<trds_in_run)
12     assert(deadlock_wait);
13     assume(!deadlock_wait
14          || cond_lock_field(*c)==0);
15     c_wait = c_wait - 1
16   }
17   mutex_lock_field(*m) = 1
18   return 0;
19 }
```

FIGURE 4.18: Modelling conditional waiting operation.

To model signal-operations, we simply release the condition variable, i.e., $c = 0$. To model broadcast-operations, we create a global variable called *broadcast_id*, which records the number of broadcast operations that have executed and which gets incremented inside *pthread_cond_broadcast*. In the wait-operation, the thread records the current *broadcast_id* before it is forced to make context switches to other threads. When the context is switched back to the current thread, an assertion checks if a broadcast operation has occurred by checking whether the current value of *broadcast_id* is greater than the recorded value. The deadlock is detected if there is no path with broadcast operations.

As example, Figure 4.19 shows a code fragment extracted again from the INSPECT suite [182], which implements the producer-consumer (also known as the bounded-buffer) application. This example consists of two threads *producer* and *consumer*, which share a global variable *num*; the *producer* increments the variable *num* (see line 12) while the *consumer* decrements it (see line 23). This simulates the access to a shared bounded-buffer. In order to make sure that the *producer* will not increment the variable *num* if it is greater than zero (i.e., try to put more data into a "buffer" that is full) and the consumer will not decrement the variable *num* if it is zero (i.e., try to remove data from a "buffer" that is empty), we use the condition variables *full* and *empty* to synchronize the *producer* and *consumer* threads. Whenever the "buffer" is full, the *producer* waits on the condition variable *empty* (see line 11); similarly, whenever the buffer is "empty", the *consumer* waits on the condition variable *full* (see line 22).

The code shown in Figure 4.19 tries to implement the producer-consumer scenario as described above, but it contains a deadlock because we initialize the variable *num* (see

```c
1  #include <pthread.h>
2  #define N 2
3  int num;
4  pthread_mutex_t m;
5  pthread_cond_t empty, full;
6  void* producer(void* arg) {
7    int i = 0;
8    while (i < N) {
9      pthread_mutex_lock(&m);
10     while (num > 0)
11       pthread_cond_wait(&empty, &m);
12     num++; //produce
13     pthread_mutex_unlock(&m);
14     pthread_cond_signal(&full);
15     i++;
16 } }
17 void* consumer(void* arg) {
18   int j = 0;
19   while (j < N){
20     pthread_mutex_lock(&m);
21     while (num == 0)
22       pthread_cond_wait(&full, &m);
23     num--; //consume
24     pthread_mutex_unlock(&m);
25     pthread_cond_signal(&empty);
26     j++;
27 } }
28 int main() {
29   pthread_t  id1, id2;
30   num = 2; //wrong initialization
31   pthread_mutex_init(&m, 0);
32   pthread_cond_init(&empty, 0);
33   pthread_cond_init(&full, 0);
34   pthread_create(&id1, 0, producer, 0);
35   pthread_create(&id2, 0, consumer, 0);
36   pthread_join(id1, 0);
37   pthread_join(id2, 0);
38   return 0;
39 }
```

Figure 4.19: An example of deadlock with condition variable on a producer and consumer application.

line 30 of Figure 4.19) incorrectly so that the thread *consumer* decrements *num* twice while the thread *producer* increments it only once. Since we have two threads and one locked (i.e., $trds_in_run = 2$ and $c_lock = 1$), our approach detects the deadlock when the exploration of the thread *consumer* terminates and the invariant $c_wait < trds_in_run$ thus becomes *false* since the *producer* is still waiting for the *consumer* to decrement the variable *num*.

4.6 Experimental Evaluation

We have implemented the lazy, schedule recording, and UW approaches described in Section 4.3 in ESBMC. In our experiments, we have used ESBMC v1.15.1 together with Z3 v2.11 [57], which was the most efficient SMT solver in our previous experiments [53] (see also Chapter 3).

The experimental evaluation of our work consists of three parts. In Section 4.6.1, we compare our approaches against the Monotonic Partial Order Reduction (MPOR) [103] and Peephole Partial Order Reduction (PPOR) [178] that are implemented in an SMT-based bounded model checker using the Yices SMT solver [65]. In Section 4.6.2, we compare our lazy approach against CHESS v0.1.30626.0 [136, 137], which is a concurrency testing tool for C# programs. CHESS supports iterative context-bounding by exploring the various thread schedules deterministically. In Section 4.6.3, we compare our approaches against SATABS version 2.5 [44] connected to Cadence SMV [124], which is a state-of-the-art C model checker and supports the verification of multi-threaded software with shared variables using the CEGAR technique.

All experiments were conducted on an otherwise idle Intel Pentium Dual CPU, 2GHz and 3GHz with 4 GB of RAM running Windows and Linux OS respectively. For all benchmarks, the time limit has been set to 3600 seconds to check all properties at once. All times given are wall clock time in seconds as measured by the unix *time* command through a single execution. In our experiments, we chose CHESS [136, 138] and SATABS [44] as two of the most widely used verification tools.

4.6.1 Comparison to MPOR and PPOR

We use the dining philosophers model to evaluate our approaches against MPOR and PPOR. MPOR and PPOR combine dynamic partial order reduction [68] with symbolic state space exploration for model checking multi-threaded software. Both MPOR and PPOR thus explore all necessary interleavings by dynamically tracking interactions between the threads interleavings and adding constraints to allow automatic pruning of redundant interleavings in the SMT solver. However, MPOR is based on the notion of quasi-monotonic sequences of thread-ids, i.e., if all transitions enabled at a global state are independent then MPOR needs to explore just one interleaving, which is chosen to be the one in which transitions are executed in increasing (monotonic) order of their thread-ids; while PPOR is based on the notion of guarded independent transitions, i.e., transitions that can be considered as independent in certain execution paths. MPOR is optimal (i.e., remove all redundant interleavings) for multi-threaded programs with more than two threads while PPOR is optimal for programs with two threads.

Since the benchmarks used by Kahlon et al. [103] are not available, we re-implemented

					MPOR	PPOR	Lazy		Sched.	UW	
	Module	L	T	B	Time	Time	Time	#FI/#I	Time	Time	Itr
1	dp2_unsat	63	2	3	0.2 (0.2)	**0.1** (0.1)	0.2	0/2	0.2	0.2	1
2	dp3_unsat	63	3	4	0.8 (0.9)	1.0 (1.1)	**0.3**	0/6	**0.3**	**0.3**	1
3	dp4_unsat	63	4	5	5.0 (5.3)	41.9 (44.9)	1.3	0/24	**1**	**1**	1
4	dp5_unsat	63	5	6	21.4 (22.9)	138.7 (148.6)	6	0/120	**5.3**	**5.3**	1
5	dp6_unsat	63	6	7	48.8 (52.3)	470.4 (504.4)	**45**	0/720	65.3	64.2	1
6	dp7_unsat	63	7	8	**150.8** (161.6)	TO -	360	0/5040	MO	MO	0
7	dp2_sat	63	2	3	0.1 (0.1)	0.1 7(0.1)	0.1	2/2	0.2	0.2	3
8	dp3_sat	63	3	4	1.2 (1.3)	0.3 (0.3)	**0.1**	6/6	0.2	0.5	3
9	dp4_sat	63	4	5	8.9 (9.5)	3.6 (3.8)	**0.2**	24/24	0.6	2.6	4
10	dp5_sat	63	5	6	88.4 (94.7)	57.6 (61.7)	**0.3**	120/120	2.8	24.5	5
11	dp6_sat	63	6	7	294.4 (315.4)	2130.8 (2283)	**0.3**	720/720	24.5	568.2	6
12	dp7_sat	63	7	8	1136.8 (1218)	TO -	**0.3**	5040/5040	818.1	816.2	1

TABLE 4.2: Results of the comparison between MPOR and PPOR, and lazy, schedule, and UW ESBMC

them as described there. The implementation is available for downloading at the ESBMC webpage (`http://users.ecs.soton.ac.uk/lcc08r/esbmc`). Each philosopher has its own local variables, and they communicate only through a global shared array of forks. This version guarantees the absence of deadlocks. As in [103], we also check two properties: *(i)* whether all philosophers can eat simultaneously (this property does not hold, i.e., the verification condition is unsatisfiable) and *(ii)* whether all philosophers have eaten at least once (this property holds, i.e., the verification condition is satisfiable). Kahlon et al. [103] run their experiments on a workstation with 2.8 GHz Xeon processor and 4GB of RAM memory running Linux OS. In order to make the results comparable, we scale their times in Table 4.2. We give both original (in brackets) and scaled timings.

Table 4.2 shows the detailed results of the comparison between MPOR, PPOR, and the three ESBMC approaches. The first column L gives the number of lines of code, the second column T reports the total number of threads and the third column B provides the unwinding bound. The *Time* column provides the time in seconds while the column #I provides the total number of generated interleavings and the column #FI the total number of failed interleavings. The column *Itr* gives the number of iterations to prove or disprove the property in the UW approach.

As we can see in Table 4.2, our approaches perform equivalently to MPOR to check the first property of the model until we increase the number of philosophers to 6. If we continue increasing the number of philosophers, MPOR performs better than our approaches. However, our three approaches perform better than PPOR in checking the first property. In addition, our lazy ESBMC scales significantly better than the other approaches (including our UW and schedule recording approaches) in checking the second (violated) property of the dining philosophers model, i.e., whether all philosophers have

eaten at least once. We also show in column #FI/#I that all interleavings generated by our lazy ESBMC are satisfiable, i.e., that each interleaving exhibits the error. In summary, our lazy approach outperforms both MPOR and PPOR for those benchmarks that generate satisfiable formulae and is still comparable to MPOR and PPOR when the generated formulae are unsatisfiable.

4.6.2 Comparison to CHESS

CHESS is a concurrency testing tool for C# programs. It implements iterative context-bounding and explores the various thread schedules deterministically [136, 138]. CHESS requires idempotent unit tests that it repeatedly executes in a loop, exploring a different interleaving on each iteration. In this respect, it is similar to our lazy approach; however, CHESS is a purely dynamic, test-based tool and originally employed a stateless search technique, although its latest version (v0.1.30626.0) performs state hashing based on happens-before graph to avoid exploring the same state redundantly.

Table 4.3 shows the detailed results of the comparison between ESBMC and CHESS on a 2GHz machine. *reorder*, *twostage*, and *wronglock* are different versions of a reader/writer program [156]. The numbers (x, y) indicate that we have x instance(s) of thread t_{set} and y instance(s) of thread t_{reader}. According to [156], increasing the number of instances of a given thread while keeping constant the number of instances of the other thread, substantially increases the "semantic hardness" of the error discovery. Note that all these benchmarks only check for a single, violated property. *micro* is a synthetic micro-benchmark [74] (shown in Figure 1.1 of Chapter 1) which checks a single valid property. It is used to check the scalability of multi-threaded software verification tools. The number in brackets indicates the total number of visible statements on each thread. In the table, L is the size of the code (in lines), and T the total number of threads. B is the number of BMC unrolling steps for each loop, while C is the context switch bound. Except for *reorder_6_bad*, C is set to the minimum number of context switches required to expose the error. We increase further the number of context switches for *reorder_6_bad* because we want to check the scalability of both tools. *Time* is the time in seconds until the error is found; timeouts are denoted by TO. For ESBMC, I is the total number of generated interleavings, while FI is the total number of failed interleavings. The column *itr* gives the number of iterations required to prove or disprove the property in the UW approach. For CHESS, *Tests* reports the approximate number of tests executed, which is not related to the number of interleavings. Both tools identify the property violation (resp. confirm that it holds) in all cases where they do not run out of time or memory.

As we can see in Table 4.3, CHESS is effective for programs where there are a small number of threads, but it does not scale that well and consistently runs out of time when we increase the number of threads. In general, CHESS times out when the number of threads increases beyond six. The relatively poor scalability of CHESS has already been

observed by [156]. In contrast, our lazy algorithm is able to find bugs quickly even when we increase the number of threads and the context bound, and consistently outperforms CHESS as well as the schedule recording and UW approaches. However, note that our lazy algorithm runs out of memory for test cases 16 and 18 when we increase the number of context switches to 18 and 13 respectively.

4.6.3 Comparison to SATABS

SATABS is an ANSI-C model checker which supports the verification of multi-threaded software with shared variables using the CEGAR technique. We compare our approaches against SATABS v2.5 [44] based on Cadence SMV using a number of multi-threaded programs taken from standard benchmark suites. Table 4.4 shows the results achieved on a 3Ghz machine. Programs that end on "bad" contain an error (i.e., at least one of the properties is satisfiable) while those that end on "ok" are correct. Here, $\#P$ gives the number of properties to be verified for each program, which includes array bounds, pointer safety, division by zero, deadlock and order violations checks. A context bound of ∞ means that we did not specify a bound. A "-" result indicates that the tool failed with an error such as internal (†) and refinement (RF) failure, memory overflow (MO), time-out (TO), or failed to detect errors in the program. A "+" indicates that the tool detected the error or proved all VCs.

Programs 1-6 are concurrent implementations of stack, queue, and circular buffer data structures; programs 1 and 2 are extracted from an embedded application [51]. Programs 7-14 are from the INSPECT benchmark [182] and use mutex and condition synchronization primitives from the Pthread library. Programs 15-17 are from the VV-lab benchmarks [156] and contain common concurrency bugs such as data races, atomicity and order violations. Programs 18-20 are embedded applications that run on a dual core processor; they are implemented in a commercial set-top box product from NXP semiconductors [141]. Program 21 is the same synthetic micro-benchmark described in Section 4.6.2, but here we increase further the number of context switches to check the scalability of our approaches.

As we can see in Table 3.3, SATABS produces refinement failures (RF) and fails with internal errors (†) for most programs. These programs contain linear arithmetic operations with arrays and the predicate abstraction technique implemented in SATABS seems to suffer from a lack of precision when dealing with arrays. However, the ability of a verification tool to check such programs is particularly important as many real-world multi-threaded programs belong to this class. SATABS also times out for large programs or for programs with many threads (cf. programs 7, 8, 9, 13, and 21). Additionally, SATABS gives false positives on programs 14-16, which contain known bugs related to data races, atomicity and order violations.

	Test Program	#L	#T	B	C	CHESS Time	CHESS Tests	Lazy Time	Lazy #FI / #I	Schedule Time	UW Time	UW Itr
1	reorder_3_bad (2,1)	84	3	3	4	1	200	<**1**	1/29	<**1**	<**1**	4
2	reorder_4_bad (3,1)	84	4	4	5	98	13000	<**1**	1/82	1	4	5
3	reorder_5_bad (4,1)	84	5	5	6	TO	429000	<**1**	1/277	4	18	6
4	reorder_6_bad (5,1)	84	6	6	7	TO	396000	<**1**	1/853	36	72	7
5	reorder_6_bad (5,1)	84	6	6	8	TO	371000	<**1**	1/2810	225	592	7
6	reorder_6_bad (5,1)	84	6	6	9	TO	367000	<**1**	1/8124	MO	MO	1
7	twostage_3_bad (2,1)	128	3	3	4	4	500	**1**	1/35	**1**	3	5
8	twostage_4_bad (3,1)	128	4	4	4	215	27000	2	1/42	**1**	4	5
9	twostage_5_bad (4,1)	128	5	5	4	TO	384000	2	1/44	**1**	5	5
10	twostage_6_bad (5,1)	128	6	6	4	TO	366000	**2**	1/45	**2**	5	5
11	wronglock_4_bad (1,3)	110	4	4	8	21	3000	**5**	2/489	10	89	9
12	wronglock_5_bad (1,4)	110	5	5	8	724	93000	**10**	3/2869	50	408	9
13	wronglock_6_bad (1,5)	110	6	6	8	TO	356000	**18**	4/12106	225	2060	9
14	wronglock_7_bad (1,6)	110	7	7	8	TO	330000	**34**	5/39100	MO	MO	1
15	micro_2_ok (100)	247	2	1	2	316	35855	<1	0/4	<**1**	<**1**	1
16	micro_2_ok (100)	247	2	1	17	TO	400000	**1095**	0/131072	MO	MO	1
17	micro_3_ok (100)	365	3	1	2	TO	272000	<1	0/9	<**1**	<**1**	1
18	micro_3_ok (100)	365	3	1	12	TO	290000	**1021**	0/121393	MO	MO	1

Table 4.3: Results of the comparison between ESBMC (v1.15.1) and Microsoft CHESS (v0.1.30626.0).

	Test Program	L	T	P	B	C	SATABS Time	SATABS Result	Lazy Time	Lazy Result	Lazy #FI / #I	Schedule Time	Schedule Result	UW Time	UW Result	UW Itr
1	circular_buffer_ok [51]	111	2	9	8	∞	†	–	**477**	+	0/12870	MO	–	MO	–	1
2	circular_buffer_bad [51]	109	2	8	8	5	†	–	<**1**	+	3/32	2	+	11	+	6
3	queue_ok [55]	147	2	12	41	∞	RF	–	**3**	+	0/6	**3**	+	**3**	+	1
4	queue_bad [55]	153	2	15	41	8	†	–	**3**	+	91/256	50	+	373	+	7
5	stack_ok [55]	105	2	5	11	12	†	–	**225**	+	0/4094	1026	+	1097	+	1
6	stack_bad [55]	106	2	6	11	4	RF	–	<**1**	+	4/16	**2**	+	6	+	4
7	fsbench_ok [182]	81	26	47	26	2	†	–	**252**	+	0/676	304	+	301	+	1
8	fsbench_bad [182]	80	27	48	27	2	†	–	<**1**	+	729/729	360	+	786	+	2
9	indexer_ok [182]	77	13	21	129	4	TO	–	595	+	0/17160	220	+	**218**	+	1
10	stateful20_ok [182]	60	2	3	20	10	†	–	**95**	+	0/1024	487	+	518	+	1
11	sync02_ok [182]	74	2	6	21	21	RF	–	**44**	+	0/121	60	+	60	+	1
12	sync02_bad [182]	74	2	6	21	21	RF	–	**8**	+	5/186	132	+	383	+	3
13	aget-0.4_bad [182]	1233	3	279	200	2	3346	+	137	+	1/1	127	+	**125**	+	1
14	bzip2smp_ok [182]	6366	3	8568	1	9	TO	–	**1800**	+	0/1294	MO	–	MO	–	1
15	reorder_10_bad (9,1) [156]	84	10	7	10	11	**1**	–	<**1**	+	1/154574	MO	–	MO	–	1
16	twostage_100_bad (99,1) [156]	128	100	13	100	4	**2**	–	88	+	1/139	93	+	195	+	5
17	wronglock_8_bad (1,7) [156]	110	8	8	8	8	**2**	–	90	+	6/104015	MO	–	MO	–	1
18	exStbHDMI_ok [141]	1060	2	24	16	20	TO	–	229	+	0/1	226	+	**213**	+	1
19	exStbLED_ok [141]	425	2	45	10	10	RF	–	**73**	+	0/11	73	+	787	+	1
20	exStbThumbs_bad [141]	1109	2	249	2	1	317	+	95	+	3/3	14	+	**12**	+	1
21	micro_10_ok (100) [74]	1171	10	10	1	17	TO	–	**254**	+	0/29260	MO	–	MO	–	1

TABLE 4.4: Results of the comparison between SATABS (v2.5) and ESBMC (v1.15.1).

Note that SATABS uses predicate abstraction and refinement, and in some sense tries to solve a harder problem than bounded model checking. However, the results in Table 4.4 indicate that this problem may still be too hard for multi-threaded applications, as SATABS is unable to prove the required properties.

We can also see in Table 3.3 that if the program contains errors at all, these errors indeed generally occur in most interleavings explored; consequently, the lazy approach is very fast for these cases. The notable exception is *wronglock_bad*, where less than 0.1% of the interleavings expose the error and SATABS is substantially faster than ESBMC (but fails to find the error); however, even here the lazy approach outperforms both the schedule recording and UW approaches. Similarly, the lazy approach is capable of handling safe programs in which the number of threads and context switches grows quickly, which makes the formula harder and often "blows up" the SMT solver. The UW approach is typically slower than schedule recording. We suspect that the proof generation of the SMT solver (which is required to produce the unsatisfiable cores) causes memory overhead and corresponding slowdowns; this was also reported previously [56].

4.7 Related Work

SMT-based BMC is gaining popularity in the formal verification community [57]. Ganai and Gupta describe a verification framework for BMC and apply several techniques to simplify the BMC problem [71]. However, the authors focus on sequential software and use only the theory of integer and real arithmetic, which does not reflect precisely the ANSI-C semantics. Armando et al. also propose a BMC approach using SMT solvers for sequential ANSI-C programs [11] by using linear arithmetic, arrays, records and restricted bit-vectors arithmetic but they do not address important constructs of the ANSI-C language.

Cimatti et al. [39] describe an approach to verify SystemC that similarly combines explicit state space exploration (i.e., the explicit exploration of the different possible interleavings) with symbolic model checking (i.e., the symbolic representation and updates of the state). However, we use BMC instead of predicate abstraction, and we implement a realistic scheduler, i.e., our scheduler may preempt a thread at any visible instruction in its execution, whereas [39] encodes the semantics of the non-preempting SystemC scheduler. We also exploit the SMT techniques on large problems by encoding all possible interleavings into a single formula.

Qadeer and Rehof present a pragmatic method to discover bugs in concurrent software in which the program analysis is restricted to executions with a bounded number of context switches [150]. However, the authors do not apply it to realistic and large concurrent software benchmarks and the integration of this context-bounded model checking algorithm

into the explicit state model checker ZING is left for future work. Rabinovitz and Grumberg describe an extension of the CBMC model checker to concurrent C programs [152], which translates C threads into SSA form and adds constraints for a bounded number of context-switches, as described in [150]. This approach, however, is limited to two threads, and requires the user to run the model checker twice in order to detect different types of bugs ("regular" and concurrency bugs). It is also only evaluated on a concurrent bubblesort, but not on a set of realistic applications.

Ganai and Gupta describe a lazy method for modelling multi-threaded concurrent systems using shared variables [73], but this method is also restricted to two threads. Gupta et al. [103] extend [73, 102] by supporting more than two threads and by combining dynamic partial order reduction with symbolic state space exploration. The benchmarks that have been reported are a parameterized version of the dining philosophers model, which are untypical multi-threaded C programs. Grumberg et al. propose an algorithmic method based on SAT and BMC to model check a multi-process system based on a series of under-approximated models [84]. This approach, however, does not integrate context-bounded analysis and it does not address the problem of model checking multi-threaded C software.

4.8 Conclusions

We have presented three different approaches to model check multi-threaded ANSI-C software with shared variable communication between the threads. The lazy approach iteratively generates all possible interleavings and calls the BMC procedure on each interleaving. The schedule recording approach systematically encodes all possible interleavings into one formula. The underapproximation and widening approach checks models with an increased set of allowed interleavings. The main contribution of all three approaches is in the combination of symbolic model checking with explicit state space exploration. As far as we are aware, the lazy approach has not been described or evaluated in the literature. Similarly, the underapproximation-widening approach has not been used for bounded model checking of multi-threaded software. The difference between our schedule recording and Gupta et al. [103] is that they work in a fully symbolic context. With these novel approaches we have successfully achieved the second objective stated in Section 1.2.

Additionally, we have presented our modelling of the synchronization primitives of the Pthread library that allows us to detect not only atomicity and order violations, but also local and global deadlock, that previous attempts are unable to find [73, 102, 103, 152]. Surprisingly, our approach to check constraints lazily is extremely fast for programs that contain errors and to a lesser extent even for safe programs in which the number of threads and context switches grows quickly. The experimental results also show

that the lazy approach generally outperforms not only the schedule recording and UW approaches, but also CHESS [136] and SATABS [44] tools on several non-trivial benchmarks. As far as we are aware, there is no other work that considers a comprehensive SMT-based BMC procedure to verify multi-threaded ANSI-C software by combining symbolic model checking with explicit state space exploration. In future work, we plan to explore the use of Craig interpolants to prove non-interference of context switches among the threads up to a given depth and develop an efficient method on top of ESBMC to localize faults in multi-threaded programs.

Chapter 5

Implementation of ESBMC

Chapter 5 describes the main software components of the ESBMC architecture. Additionally, in order to achieve the third objective stated in Section 1.2, we describe the simplifications and heuristics that we used in order to reduce the unwound formula and to determine the best representation for the program variables. It also evaluates the simplifications and heuristics, which show a substantial performance improvement over a large set of benchmarks. The results described in the previous chapters have been achieved using the implementation described here, so this chapter should not be interpreted as a continuation of the previous chapters, but a "separation of concerns".

5.1 Introduction

ESBMC is a context-bounded model checker for embedded ANSI-C software based on SMT solvers. It allows the verification engineer to:

- verify single- and multi-threaded software (with shared variables and locks);
- reason about arithmetic under- and overflow, pointer safety, memory leaks, array bounds, atomicity and order violations, deadlock, data race, and user-specified assertions;
- verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic.

ESBMC does not require the user to annotate the programs with pre/post-conditions, but allows the user to state additional properties using *assert*-statements, that are then checked as well. It also provides three approaches (lazy, schedule recording, and under-approximation and widening) to model check multi-threaded software. ESBMC can be invoked through the command-line interface or configured through the Eclipse plug-in

(see Appendix A). ESBMC converts the verification conditions using different background theories and passes them directly to an SMT solver. In addition, ESBMC can output verification conditions using the SMT logics QF_AUFBV and QF_AUFLIRA. ESBMC is built on top of the CProver framework; the next section explains ESBMC's overall architecture, and the framework modifications.

5.2 Tool Architecture

Figure 5.1 shows its main software components. Every step of the model checking process in ESBMC is implemented within a separate software component. ESBMC is written in C++ and can be executed on all major operating systems and machines (i.e., 32-bit Windows/x86, 32-bit Linux/x86, 64-bit Windows/x64 and 64-bit Linux/x64). In Figure 5.1, the white boxes (except for the SMT solver) represent the components that we reused from the CProver framework without any modification while the gray boxes with dashed lines represent the components that we modified in order to:

1. generate automatically assertions to check for memory leaks, data races, atomicity and order violations and deadlocks (implemented in the component *GOTO program*, see Subsections 3.4.7, 4.4 and 4.5);
2. extend the SSA form of the symbolic execution engine to avoid naming conflicts when verifying multi-threaded programs (implemented in the component *GOTO symex*, see Subsection 4.3.1);
3. simplify the unwound formula based on high-level information to prevent overburdening the solver (implemented in the component *GOTO symex*, see Subsections 3.3 and 5.3);
4. perform an up-front analysis in the CFG of the program to determine the best encoding and solver for a particular program (implemented in the component *GOTO symex*, see Subsection 5.4).

The *GOTO program* component converts the ANSI-C program into a *goto*-program, which simplifies the representation (e.g., replacement of *switch* and *while* by *if* and *goto* statements). The *GOTO symex* component performs a symbolic simulation of the program, which thus handles the unrolling of the loops and the elimination of recursive functions; and generates the verification conditions to be encoded in the back-end.

In Figure 5.1, the gray boxes with solid lines represent new components that we implemented from scratch in order to guide the symbolic execution via a thread scheduler (see the component *Scheduler*) and to encode the given constraints and properties of an ANSI-C program into a global logical context (see the components *constraints* and

properties), using the background theories supported by the SMT solvers. We also implemented new components from scratch to interpret the counter-example generated by the supported SMT solvers (see component *Interpret counter-example*). The software components to convert the constraints and properties and to interpret the counter-example must be implemented in the back-end to support each new SMT solver. In total, we implemented approximately 20000 lines of C++ code, which approx. 80% belong to the back-end and 20% belong to the front-end.[1]

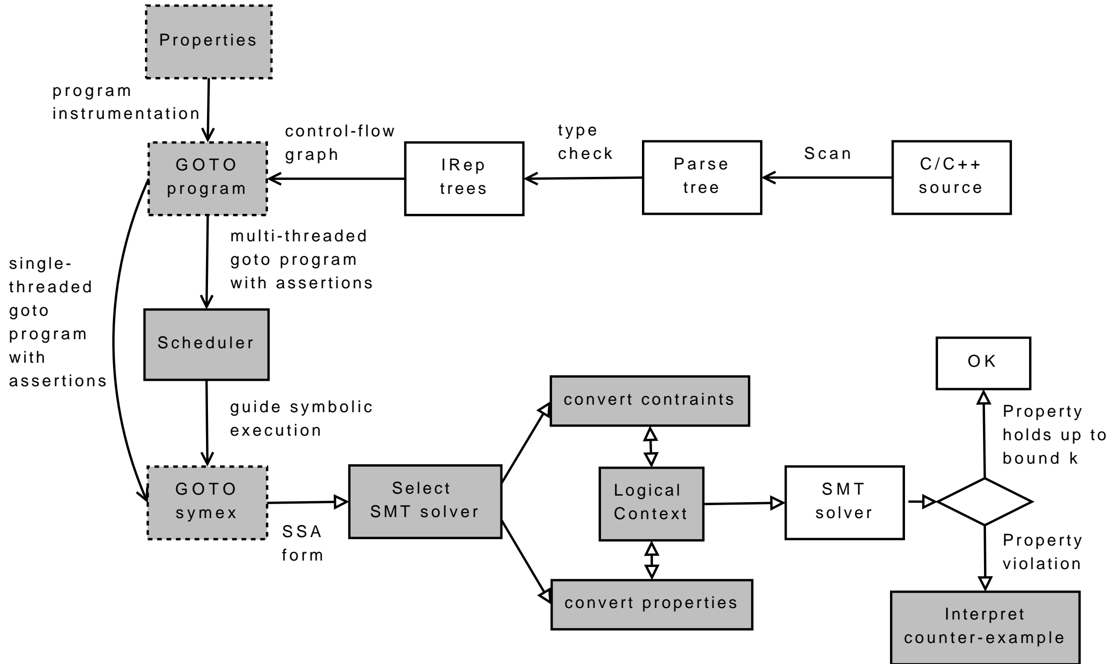

FIGURE 5.1: Overview of the ESBMC architecture.

In the back-end of ESBMC, we build two sets of quantifier-free formulae C (for the constraints) and P (for the properties) so that C encodes the first part of ψ_k (more precisely, $I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$) and $\neg P$ encodes the second part (more precisely, $\bigvee_{i=0}^{k} \neg\phi(s_i)$). After that, we check $C \models_{\mathcal{T}} P$ using an SMT solver. If the answer is satisfiable, we have found a violation of the property ϕ, which is encoded in ψ_k. If not, the property holds up to the bound k.

5.3 Code Simplification and Reduction

We observed during development and preliminary evaluation that constant propagation and forward substitution techniques [134] significantly improved the performance of ESBMC over a wide range of embedded software applications. Constant propagation allows us to replace the values of known constants in expressions at verification time.

[1] Measured by running `find . -name "*.[cpp]" | xargs wc -l` and `find . -name "*.[h]" | xargs wc -l` in the project directory.

The front-end of ESBMC, in particular the *GOTO symex* component, already propagates constants related to scalar variables, but not for pointers to objects and for store operations that update the content of arrays, structs and unions. Figure 5.2 shows a fragment extracted from the cyclic redundancy check algorithm of the SNU-RT benchmark [116] as an example where constant propagation for scalar variables leads to a significant performance improvement.

```
1  ...
2  int icrc1(int crc, int onech);
3  static int icrctb[256], rchr[256];
4  static int it[16]={0,8,4,12,2,10,6,14,1,9,5,
5                     13,3,11,7,15};
6  int j;
7  for (j=0;j<=255;j++) {
8    icrctb[j]=icrc1(j<<8,0);
9    rchr[j]=(it[j&0xF]<<4 | it[j>>4]);
10 }
11 return 0;
12 ...
```

FIGURE 5.2: Code fragment of cyclic redundancy check.

Figure 5.3 shows the *goto*-program for the code fragment in Figure 5.2, which adds additional assertions to check for array bounds violation. As explained in Section 3.4.4, the verification conditions to check for array bounds violation do not require the array theory. Therefore, after unwinding the loop of the *goto*-program shown in Figure 5.3, the validity of the array bounds check (see lines 3, 5, 6 and 7) can be evaluated statically (e.g., by simply propagating the value of the scalar variable j during the loop unwinding). Figure 5.4 shows the loop unwinding of the *goto*-program shown in Figure 5.3. Note that the expressions $icrctb1 = ARRAY_OF(0)$ and $rchr1 = ARRAY_OF(0)$ mean that all elements of the arrays *icrctb1* and *rchr1* are initialized to zero.

```
1  ...
2  1: if !(j<=255) then goto 2
3     assert j<256 //array 'icrctb' upper bound
4     icrctb[j]=icrc1((j<<8),0)
5     assert j<256 //array 'rchr' upper bound
6     assert (15&j)<16 //array 'it' upper bound
7     assert j>>4<16 //array 'it' upper bound
8     rchr[j]=((it[j&15])<<4 | (it[j>>4]));
9     j = j + 1;
10    goto 1
11 2: return 0;
12 ...
```

FIGURE 5.3: *Goto*-program for the code fragment in Figure 5.2.

We also exploit the constant propagation technique to replace pointers to objects that are constants by the respective constant and to replace store operations that update the content of arrays, structs and unions with constant values by the values of the

```
1  icrctb1 == ARRAY_OF(0)
2  rchr1 == ARRAY_OF(0)
3  it1 == { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
4           13, 3, 11, 7, 15 }
5  j1 == 0
6  icrctb2 == (icrctb1 WITH [0:=0])
7  rchr2 == (rchr1 WITH [0:=0])
8  j2 == 1
9  icrctb3 == (icrctb2 WITH [1:=4129])
10 rchr3 == (rchr2 WITH [1:=128])
11 ...
12 j3 == 255
13 icrctb257 == (icrctb256 WITH [255:=7920])
14 rchr257 == (rchr256 WITH [255:=255])
```

FIGURE 5.4: Loop unwound for the *goto*-program in Figure 5.3.

known constants. Figure 5.5 shows an example extracted from the POWERSTONE benchmark [159] to illustrate how constant propagation works for pointers in ESBMC.

```
1 ...
2 void puts(const char *s) {
3   while( *s ) {
4     putc(*s++);
5   }
6 }
7 ...
8 puts("blit: success");
```

FIGURE 5.5: Code fragment of blit.

The function *puts* defined in line 2 is often called with a pointer to an array of constants (e.g., see line 8), but CBMC's VCG still generates VCs to check for the bounds of the pointer s as explained in Section 3.4.6. During the unrolling phase, we check whether the last value assigned to a pointer is a constant, and if so, we replace it by the constant and pass the modified expression to a simplifier, which is able to perform simple deductions before generating a VC to be encoded by the back-end.

We also propagate the *store*-operations for arrays, structs and unions up to a certain level. Figure 5.6 shows an example extracted from the EUREKA benchmark [148] to illustrate how constant propagation works for arrays. In line 2, we initialize the first position of array a with a constant (see line 2). In each iteration of the *for*-loop, we add the value of the loop counter i to the last value written in array a and write the result to the next position of a (line 4). After the loop, we then check whether the assertion in line 6 holds. However, after unrolling the loop N times we obtain a large expression $store(\ldots(store(store(store(a_0, 0, 1), 1, 2), 2, 4), 3, 7), \ldots)$ of nested store operations for a. Since all arguments except a_0 are constants, we can in principle check statically whether the assertion in line 6 holds. In practice, however, the model checker becomes slower than the SMT solvers in propagating these constants if the expressions become too large. In

our benchmarks, we observed a substantial improvement in performance if we propagate the known constants up to six nested store operations. We thus reduce substantially the number of VCs, but we leave the harder cases for the SMT solvers.

```
1 ...
2 a[0]=1;
3 for( i=1; i<N; i++){
4   a[i]= a[i-1] + i;
5 }
6 assert(a[i-1]<2*1000);
7 ...
```

FIGURE 5.6: Code fragment of SumArray.

We also observed that several applications repeat the same expression at many different places, especially after loop unrolling, in a way that the value of the operands does not change in between the occurrences. This can be detected easily in the SSA form and used for caching and forward substitution. Figure 5.7 shows a fragment of the Fast Fourier Transform (FFT) algorithm, extracted again from the SNU-RT benchmark [116], as an example of where the forward substitution technique can be applied. This occurs because the SSA representations of the two outermost *for*-loops (in lines 6-15 and lines 8-14, respectively) will eventually contain several copies of the innermost *for*-loop (lines 10-13), and thus the right-hand side of the assignment in line 11 is repeated several times in the SSA form, depending on the unwinding bound used to model check this program.

```
1  typedef struct {
2    float real, imag;
3  } complex;
4  int n=1024;
5  complex x[1024], *xi;
6  for(le=n/2; le>0; le/=2) {
7    ...
8    for (j=0; j<le; j++) {
9      ...
10     for (i=j; i<n; i=i+2*le) {
11       xi = x + i;
12       ...
13     }
14   }
15 }
```

FIGURE 5.7: Code fragment of Fast Fourier Transformation.

For example, if we set the unwinding bound k to 1024 (which is required because the upper bound n of the innermost *for*-loop is equal to 1024, see line 4), the *for*-loop in lines 6-15 will contain nine copies of the *for*-loop in lines 8-14, where the variable le will assume the values 512, 256, 128, ..., 1. Consequently, the expression $x + i$ that is assigned to the xi pointer index is replaced up to nine times for each value that i takes in the *for*-loop in lines 10-13. We thus include all expressions into a cache so that when

a given expression is processed again in the program, we only retrieve it from the cache instead of creating a new copy using a new set of variables.

We also try to simplify the quantifier-free formulae C and P by using local and recursive transformations in order to remove functionally redundant expressions and redundant literals as follows:

$$\begin{array}{ll} a \wedge true = a & a \wedge false = false \\ a \vee false = a & a \vee true = true \\ a \oplus false = a & a \oplus true = \neg a \\ ite\,(true, a, b) = a & ite\,(false, a, b) = b \\ ite\,(f, a, a) = a & ite\,(f, f \wedge a, b) = ite\,(f, a, b) \end{array}$$

We apply these simplifications to try reducing the size of the unrolled formula and consequently achieve simplification not only within each time step but also across time steps during the unwinding of the program.

5.4 Exploiting Datatype Representations

As mentioned in Chapter 3, modern SMT solvers provide ways to model the program variables either as bit-vectors or as elements of an abstract numerical domain (e.g., $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$). If the program variables are modelled as bit-vectors of fixed size, then the result of the analysis can be precise (w.r.t. the ANSI-C semantics), depending on the size considered for the bit-vectors. On the other hand, if the program variables are modelled as numerical values, then the result of the analysis is independent from the actual binary representation, but it may not be precise when arithmetic expressions are involved. As a motivating example, consider the following small C program from [42] as shown in Figure 5.8.

```
1  int main() {
2    unsigned char a, b;
3    unsigned int result=0, i;
4    a=nondet_uchar();
5    b=nondet_uchar();
6    for(i=0; i<8; i++)
7      if((b>>i)&1)
8        result+=(a<<i);
9    assert(result==a*b);
10 }
```

FIGURE 5.8: A C program that uses shift-and-add to multiply two numbers.

This program non-deterministically selects two values of type *unsigned char* and uses bitwise AND, right- and left-shift operations to multiply them. Reasoning about this

program by means of integer arithmetic produces wrong results if the bit-level operators are treated as uninterpreted functions (UFs) because, even though UFs simplify the proofs, they ignore the semantics of the operators and consequently make the formula weaker. This problem occurs in several software model checkers (e.g., SMT-CBMC [11] handles restricted bit-vectors arithmetic and BLAST [88] treats bit-level operations as UFs, models integers as elements of $\mathbb{Z}$ and does not account for arithmetic overflows [21]), which fail to check the assertion in line 9. In contrast, bit-vector arithmetic allows us to encode bit-level operators in a more accurate way. However, in our benchmarks, we noted that the majority of VCs are solved faster if we model the basic datatypes as $\mathbb{Z}$ and $\mathbb{R}$. Consequently, we have to trade off between *speed* and *accuracy* which are two competing goals in formal verification using SMT.

Based on the extent to which the SMT solvers support the domain theories and on experimental results obtained with a large set of benchmarks, we developed a simple but effective heuristic to determine the best representation for the program variables as well as the best SMT solver to be used in order to check the properties of a given ANSI-C program:

1. Our default representation for encoding the constraints and properties of a given ANSI-C program are integers and reals, respectively, and our default SMT solver is Z3.
2. We then explore the CFG representation of the program.
3. If we find expressions that involve bit-level operations (e.g., $<<$, $>>$, $\&$,$|$, $\oplus$) or typecasts from signed to unsigned datatypes and vice-versa, we encode the corresponding variables as bit-vectors.
4. We switch the SMT solver to Boolector if no pointers are used but we keep Z3 if pointers are used.

We adopted this strategy because we are able to implement the theory of tuples on top of Z3 to model pointers and thus exploit the structure provided by the word-level instead of bit-level models (i.e., instead of concatenating and extracting bit-vectors) [108].

5.5 Evaluation of Performance Improvements

We evaluate the effectiveness of the simplification techniques and the exploration of the datatype representations described in Sections 5.3 and 5.4 resp. using 174 programs, with a total size of 70K lines of code, taken as a representative sample from the benchmark suites Siemens, SNU-RT, PowerStone, NECLA and NXP. With all optimizations enabled, ESBMC can check all 174 programs in 439 seconds, which serves as our baseline.

We then evaluate the effect of the simplifications by disabling them one at a time as follows: constant propagation of store operations for arrays, structs and unions (CP_{store}); constant propagation for constant strings (CP_{String}); forward substitution (FS); and removal of functionally redundant literals and variables ($FRLV$). We set the time out to 180 seconds because this is longest time to check a given program with all optimizations enabled.

Surprisingly, ESBMC performs marginally better when we disable the removal of functionally redundant literals and variables ($FRLV$) and checks all 174 programs in 423 seconds. We can thus conclude that the SMT solvers already eliminate the functionally redundant literals and variables during the preprocessing phase in a more efficient way. Fortunately, all other simplifications pay off. Using CP_{store}, ESBMC checks 170 programs in 1059 seconds and times out in four programs. With CP_{String}, it checks 173 programs in 590s and times out in one program, and with FS, it checks 171 programs in 972 seconds and times out in three programs. The optimizations are complementary in the sense that disabling each one of them causes ESBMC to time out on different programs. Moreover, their effect is not only restricted to the programs that ESBMC fails to check when they are disabled: on the remaining 166 programs, disabling CP_{store} causes an average slow-down of more than 30%. However the effects are less pronounced, or even reversed, for disabling CP_{String} and FS, with a slow-down of approx. 8%, and a speed-up of approx. 4%, respectively.

We also evaluated the effect of automatically selecting the best representation for the program variables using the SMT logics QF_AUFBV and QF_AUFLIRA together with the SMT solver Z3 in order to encode the verification conditions using all simplifications described above. If we use QF_AUFBV (i.e., the bit-vector representation), ESBMC checks 170 programs in 1143 seconds and times out in four programs; and it does not give any false (positive or negative) result. If we use QF_AUFLIRA (i.e., integers and reals), ESBMC checks all 174 programs in 419 seconds, but it gives a false negative in one program in which the property to be checked contains bit-level operators (& and $\oplus$) and typecasts from unsigned to signed integer (note that this false result does not show up in our experiments in Chapters 3 and 4 because we used the heuristics described in Section 5.4). Differently from other simplifications, their effect is restricted to the programs that ESBMC fails: using our heuristics as described in Section 5.4, ESBMC checks the remaining 170 programs in 336 seconds, encoding the VCs using QF_AUFBV causes an average slow-down of 1% while encoding the VCs using QF_AUFLIRA gives a speed-up of approx. 4%.

5.6 Conclusions

We presented the main software components of the ESBMC architecture, the simplifications that we applied to reduce the unwound formula and the heuristics that we used to determine the best representation for the program variables. With the implementation of these simplifications and heuristics we successfully achieved the third objective stated in Section 1.2. Moreover, we have seen that every step of the model checking process in ESBMC is implemented within a separate component. The communication between the software components is conducted by means of well-defined interfaces. Therefore, single components of the model checking process in ESBMC could, in principle, be exchanged independently. We have also shown that the simplifications CP_{store}, CP_{String} and FS reduce substantially the unwound formula that is passed to the SMT solvers. Additionally, we also observed in our benchmarks that the majority of the verification conditions are solved faster if we model the basic datatypes as integer and/or real as specified in the SMT-LIB.

Chapter 6

Integrating ESBMC into Software Engineering Practice

In this chapter, we describe an approach to integrate SMT-based bounded model checking into the software engineering process by exploiting practices such as incremental development and regression tests; this chapter is directed towards the fourth objective stated in Section 1.2. In particular, our approach looks at the modifications suffered by the software system since its last verification, and submits them to a partly static, partly dynamic "continuous" verification process, guided by a set of test cases for coverage. A case study from the telecommunications domain shows that the proposed approach can potentially improve the error-detection capability and reduce the overall verification time.

6.1 Introduction

The complexity of software in embedded systems has increased significantly over the last years so that software verification now plays an important role in ensuring the overall product quality. In this context, bounded model checking has been successfully applied to discover subtle errors, but for larger applications, it often suffers from the state space explosion problem, as we pointed out in Chapter 3. We try to address this bottleneck with a new concept called *continuous verification*, which combines existing ideas of software engineering (e.g., continuous integration [70]) and formal verification (e.g., equivalence checking [30]) communities.

The continuous verification approach thus aims to automatically detect design errors and integration problems as quickly as possible by exploiting information from the software configuration management (SCM) system, systematically focusing the verification effort on new or modified functions. We use equivalence checking to determine whether

modified functions need to be re-verified formally and we use existing test cases to reduce the search space for the model checker, thus combining dynamic and static verification.[1]

The formal verification community has extensively used equivalence checking for hardware designs [30, 109], but there is little evidence that equivalence checking for large embedded software will improve the scalability of software model checking. In particular, Godlin and Strichman describe an approach to prove the equivalence of similar programs [78, 167] and apply it to random and industrial programs (e.g., ranging from 300 to 3000 lines of code). The authors claim that their approach takes from few seconds to 30 minutes in order to prove equivalence on equivalent programs or it can take several hours (or run out of memory) on non-equivalent programs. The results are inconclusive since Godlin and Strichman do not specify how many functions they are able to prove in their benchmarks, the time needed to check each one, how many functions are actually equivalent and how often these functions are modified from one version to another. Matsumoto et al. also describe an approach to check the equivalence of C programs using the SMT solver CVC, but the authors restrict the C programs to be checked (e.g., no pointer uses) due to limitations of their symbolic execution engine [121]. The paper also does not provide sufficient details to compare their results to the results of our approach.

The main purpose of this chapter is thus to investigate whether the continuous verification approach can indeed substantially reduce the verification time of large embedded software using our SMT-based context-bounded model checker.

6.2 Continuous Verification

The continuous verification approach has its roots in the continuous integration (CI) practice described by Fowler [70]. CI relies on every developer to create and execute unit, functional and integration tests before committing their source code to a single source repository. It also assumes the existence of an automated unit test framework. The SCM is then used to perform the system build and test processes in a completely automatic way. In *continuous verification*, we use the same information (i.e., development history and test cases), but in a different way to improve the coverage and substantially reduce the verification time throughout the development of a product or product line. We use SMT-based bounded model checking to verify for each system build that the entire system still satisfies all properties given as assertions by the designers, as well as a range of language-specific safety properties such as the absence of arithmetic under- and overflow, out-of-bounds array indexing, NULL-pointer dereferencing, or memory leaks. We also consider properties expressed in LTL which we can easily convert to C-monitors via Büchi automata [173] and model-check with ESBMC. Figure 6.1 shows the main

[1] We use the term *dynamic* to denote that the program is executed and its actual and expected outputs observed and *static* to denote that a mathematical model of the program is analyzed.

elements and steps of the continuous verification approach; the gray boxes indicate core steps.

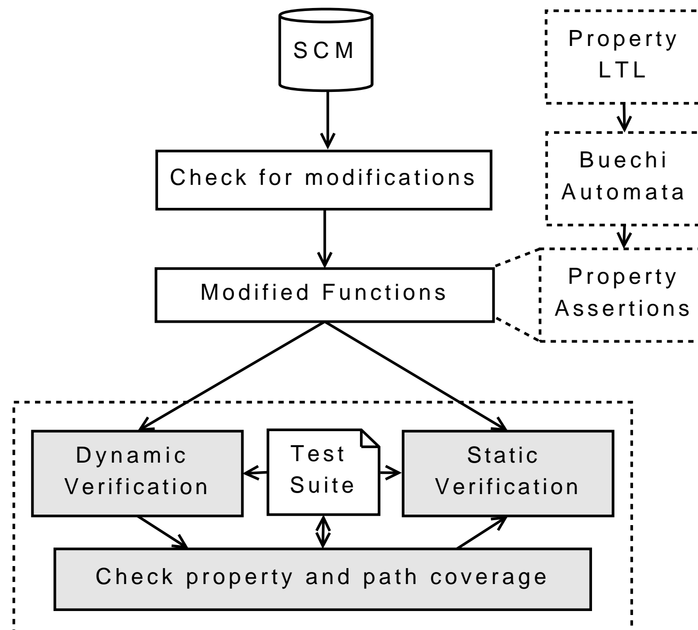

FIGURE 6.1: Continuous Verification

For large embedded software systems, the computational effort to re-verify the entire software from scratch is high, and is even largely wasted if, as is often the case, the changes are small [70]. For each system build, we thus consult the SCM to identify the functions and methods that have actually been modified and focus on these. We then use equivalence checking to determine whether they need to be re-verified formally: if we can prove that the old and new versions of a function are functionally equivalent, then we do not need to show for the new version any of the properties already shown for the old version. This can potentially reduce the immediate verification effort because proving the equivalence of two function versions can be less expensive than re-verifying the function [78, 109, 167]. However, and more importantly, it also reduces overall system verification efforts because it limits the propagation of changes through the system: if we can prove the two versions of the function computationally equivalent, then we do not need to re-verify any other function that depends it (unless that function has been changed as well). Of course, proving the equivalence of two functions is in general undecidable, due to unbounded memory usage [109], and the effort we spend in trying to do so might be wasted.

As an example, consider the two versions of the *signalInverter* function shown in Figure 6.2. They were extracted from the embedded software of two releases of a medical device product. In order to prove the equivalence of these two ANSI-C functions, we compare their input-output relations. We thus:

```
1 unsigned signalInverter(int signal) {
2   unsigned inverter;
3   if(signal >= 0)
4     inverter = signal;
5   else
6     inverter = -1*signal;
7   return inverter;
8 }
```

(a)

```
1 unsigned signalInverter(int signal) {
2   if(signal < 0)
3     return -signal;
4   else
5     return signal;
6 }
```

(b)

FIGURE 6.2: (a) Original function to invert the sign of signal. (b) Optimized version.

1. remove from each function the variable declarations and return statements;
2. convert the function bodies into single static assignment (SSA) form [134] (i.e., we introduce fresh variables by subscripting the original name such that every assignment has a unique left hand side);
3. conjoin all program statements.

These operations produce two intermediate formulas α_1 and α_2 representing the functions' computations, as shown below.

$$\alpha_1 \equiv inverter_1 = signal_1 \wedge \; inverter_2 = -1 * signal_1 \\ \wedge \; inverter_3 = (signal_1 \geq 0 \; ? \; inverter_1 \; : \; inverter_2)$$

$$\alpha_2 \equiv signal'_2 = (signal'_1 < 0 \; ? \; -signal'_1 \; : \; signal'_1)$$

Note that in general we need to rename apart the formulas, because we need to prevent that unrelated intermediate steps are accidentally related. For the actual equivalence check, we identify the input variables (i.e., $signal_1 = signal'_1$), and show using SMT-based bounded model checking that, given the representation of the function bodies, the output variables then also coincide:

$$(\alpha_1 \wedge \alpha_2 \wedge (signal_1 = signal'_1)) \Rightarrow (inverter_3 = signal'_2) \tag{6.1}$$

If the functions access a global variable g, we also have to ensure that the value of g coincide for both functions, i.e., we add the term $g_1 = g'_1$ to the consequent of the above

formula. As in [78, 167], we also abstract calls to other functions with uninterpreted function symbols with the purpose of keeping the size of the SMT formula relatively small. This is sound as long as the called functions have no side-effects and have been proved to be equivalent.

6.3 Generalizing Test Cases

After detecting new and/or modified functions, we use the existing unit test cases to reduce the state space to be explored by the model checker. In this phase, we first run the unit tests, keeping track of which inputs have already been used. We then guide the model checker to visit states that have not been visited previously (e.g., by placing assumptions on the input). In addition, the test cases also help to reduce the state space to be explored in another way: by using the test stubs, we can break the global model (containing the entire program) into local models (containing only the functions under test) and generate on-demand the reachable states to be visited by the model checker, starting with the state described by the test case. We can so reduce the number of paths and variables to be considered during model checking.

This approach is similar to *concolic testing*, which simultaneously executes a program concretely and symbolically [119, 160]. However, here we do not generate new concrete values for the test cases with the purpose of maximizing the code coverage. Instead, we use existing test cases and *assume*-statements to block larger parts of the search space (e.g., by combining respective concrete values of the test cases into a single interval).

As an example consider the three simple C functions shown in Figure 6.3 that were extracted from a medical device, and one of the test cases shown in Figure 6.4. The device, called a pulse oximeter [51], is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method. The functions, that we consider here from the pulse oximeter, implement a simple circular buffer using a FIFO (First In, First Out) policy. The test case checks whether messages are correctly added to and removed from the circular buffer using the FIFO policy. Other test cases check for buffer underflow and/or overflow and whether the elements are lost before reading them from the buffer.

The pulse oximeter sources contain seven test cases, which *intend* to cover all possible execution paths related to the circular buffer, and during dynamic verification, we are not able to find any bug in the circular buffer implementation with these. However, the implementation is flawed: the array *buffer* is declared to be of type *char*[] (see line 1 in Figure 6.3) but we assign an element *b* of type *int* (see line 14). The test cases do not uncover this error because they happen to use only integer values that can safely be cast to a *char*.

```
1  static char  buffer[BUFFER_MAX];
2  void initLog(int max) {
3    buffer_size = max;
4    first = next = 0;
5  }
6
7  int removeLogElem(void) {
8    first++;
9    return buffer[first-1];
10 }
11
12 void insertLogElem(int b) {
13   if (next < buffer_size) {
14     buffer[next] = b;
15     next = (next+1)%buffer_size;
16     assert(next<buffer_size);
17   }
18 }
```

FIGURE 6.3: Implementation of a circular buffer.

```
1  static void testCircularBuffer(void){
2    int ssr[]={1, -128, 98, 88, 59,
3                    1, -128, 90, 0, -37};
4    int i;
5    initLog(5);
6    for(i=0; i<10; i++)
7      insertLogElem(ssr[i]);
8    for(i=5; i<10; i++)
9      TEST_ASSERT_EQUAL_INT(ssr[i],
10                           removeLogElem());
11 }
```

FIGURE 6.4: A unit test for the functions shown in Figure 6.3.

Using SMT-based bounded model checking, we can detect this bug by non-determinininistically assigning a value to the parameter b (i.e., by adding an assignment $b = nondet_int()$ after line 12 in Figure 6.3). However, in general this approach can lead to false negatives because the non-deterministic choice of values for program variables may force the exploration of paths that are infeasible in the original program. Rather than modifying the program we thus modify the test stubs and replace the concrete input values by non-deterministic choices. Here, we replace the initialization of the array *ssr* (see line 2 of Figure 6.4) by $int\ ssr[] = \{nondet_int(), \ldots, nondet_int()\}$. We then use *assume*-statements to force the model checker away from the values that have already been explored during testing.

In order to block larger parts of the search space, we use the given concrete values from all stubs and combine the respective values into a single interval for each variable or array element; here we assume that all "obvious" boundary values are used in some of the stubs (e.g., using boundary-value analysis [166]), so that we force the model checker

Test Case	ssr[0]	ssr[1]	ssr[2]	ssr[3]	ssr[4]	ssr[5]	ssr[6]	ssr[7]	ssr[8]	ssr[9]
TC1	1	-128	98	88	59	1	-128	90	0	-37
TC2	43	-28	-98	18	-90	0	-1	1	0	-37
TC3	43	-28	-98	18	-90	0	-1	1	0	-37
TC4	1	-5	50	40	-20	1	-50	20	0	-37
TC5	1	10	-60	60	30	1	-10	40	0	-37

Table 6.1: Concrete values to check the circular buffer.

towards the "unobvious" errors. In the example, we thus add an *assume*-statement such as $assume(ssr[0]<1 \ \&\& \ ssr[0]>43)$ as shown in Figure 6.5 and we are then able to find two bugs related to overflow and underflow. Table 6.1 shows all concrete values to check dynamically the circular buffer and that we used to derive the single intervals (shown in Figure 6.5).

```
1  static void testCircularBuffer(void) {
2    int ssr[] = {nondet_int(),..., nondet_int()};
3    assume(ssr[0] <1 && ssr[0] > 43);
4    assume(ssr[1]<-128 && ssr[1]>-28);
5    assume(ssr[2]<-98 && ssr[2]>98);
6    assume(ssr[3]<18 && ssr[3]>88);
7    assume(ssr[4]<-90 && ssr[4]>59);
8    assume(ssr[5]<0 && ssr[5]>1);
9    assume(ssr[6]<-128 && ssr[6]>-1);
10   assume(ssr[7]<1 && ssr[7]>90);
11   assume(ssr[8]!=0);
12   assume(ssr[8]!=-37);
13   ...
14   int i;
15   initLog(5);
16   for(i=0; i<10; i++)
17     insertLogElem(ssr[i]);
18   for(i=5; i<10; i++)
19     ASSERT_EQUAL_INT(ssr[i],
20       removeLogElem());
21 }
```

Figure 6.5: The modified unit test for the test case shown in Figure 6.4.

6.4 Specifying Temporal Properties with Büchi Automata

In addition to the language-specific safety properties as described previously in Chapters 3 and 4, we can also show user-specified properties. These can be given directly as assertions in the code, using C's `assert` macro to state an assumption, or as formulas in linear-time temporal logic (LTL), which can track temporal properties of the software design. We translate the LTL formulas into Büchi automata using the Wring tool [165] and further into ANSI-C and merge them into the code. The resulting ANSI-C pro-

gram then monitors the design's progress and watches out for violations of the specified properties.

As an example, we extract two properties from the specification of the pulse oximeter device, and show how they can be modelled and used in the context of the continuous verification. In particular, we verify:

(a) the data flow to compute the HR value that is provided by the pulse oximeter sensor hardware.

(b) whether the user of the pulse oximeter is capable of adjusting the sample time of the embedded device.

The properties (a) and (b) can be expressed using the following LTL pattern (as described in Chapter 2):

$$AG\,(p \rightarrow Fr) \tag{6.2}$$

Here, A ("for all paths"), G ("always"), and F ("eventually") are the LTL quantifiers, and p and r represent the required pre- and post-states. In the example, for the property (a), p denotes the state in which the buffer contains HR and SpO_2 raw data, while r denotes the state that defines the respective HR value. Consequently, (6.2) specifies that any state containing the HR and SpO_2 raw data in the buffer is eventually followed by a state representing the respective HR value.

A Büchi automaton is a finite automaton over infinite words. It differs from a standard finite automaton over finite words in the definition of accepting a word, which is based on passing through an accepting state infinitely often (rather than terminating in a final state) [40]. The Büchi automata we consider here work over computation traces, i.e., sequences of states of the program to be analyzed. These are abstracted by the predicates of interest (here p and r). Hence the "words" can be represented by sequences of propositional expressions over the variables p and r. Figure 6.6 shows the non-deterministic Büchi automaton that represents the LTL formula (6.2) and Figure 6.7 shows its corresponding ANSI-C monitor. The transition function δ is given in Table 6.2.

	1	$r \vee \neg p$	r	$\neg p$
init	$\{S1, S2\}$	S3	init	init
S1	S1	S1	S3	S1
S2	S2	S2	S2	S3
S3	S3	S3	S3	S3

TABLE 6.2: Transition function δ for the Büchi automaton shown in Figure 6.6.

From the initial state, we can transition to S3 if $r \vee \neg p$ holds, stay in the initial state if either r, or $\neg p$ holds, or non-deterministically transition to either S1 or S2 if none of the

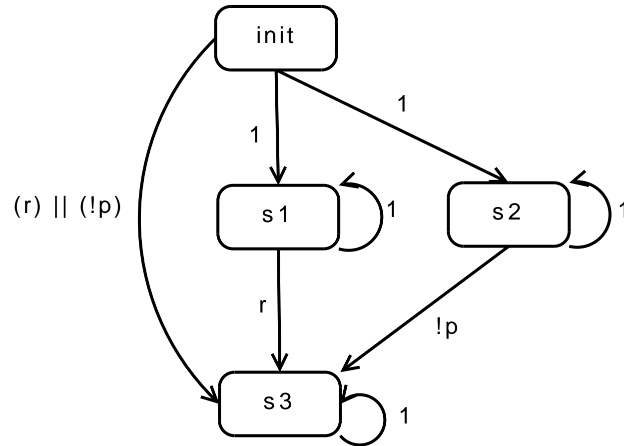

FIGURE 6.6: Specifying Temporal Properties for Software.

three properties hold (denoted by 1). This automaton will accept all infinite words that represent computations in which each state in which p holds will eventually be followed by a state in which r holds.

```
void* monitor_thread(void* arg) {
  ...
  while (1) {
    choice = nondet_uint() % 2;
    if (p) flag = true;
    switch(state) {
      case init:
        if (r || !p) state = s3;
        else if (choice == 0) state = s1;
        else state = s2;
        break;
      case s1:
        if (r) state = s3;
        else state = s1;
        break;
      case s2:
        if (!p) state = s3;
        else state = s2;
        break;
      case s3:
        state = s3;
        break;
      default:
        abort();
    }
    if (flag && !is_processing) assert(state == s3);
  }
  pthread_exit(NULL);
}
```

FIGURE 6.7: The C-monitor thread to watch out for violations of the specified property.

In order to model the non-deterministic transition of the Büchi-automata in the ANSI-

C specification, we use the function *nondet_uint*() (which returns any number of type *unsigned int* as described in Chapter 3) and then restrict its return value to the domain $\{0, 1\}$, as shown in line 4 of Figure 6.7. The property is then checked by using a "monitor" in such a way that the C program monitors the design's progress and watches out for a specific type of error up to the bound k. An assertion is then used to claim that an error is never encountered, i.e., to claim that the accepting state (S3) is reached (see line 26 of Figure 6.7). Note that we use the Boolean variable *flag* in the monitor thread to check whether p has occurred (i.e., to check whether the buffer contains HR and SpO_2 raw data).

In our example, we ensure that the buffer is not empty and contains the computed HR and SpO_2 values. The C-monitor together with the assertion is included into a thread, which interleaves with two other threads: the main thread that contains the code to be checked and an event thread that models the hardware interrupt and consequently interacts with the pulse oximeter hardware, as shown in Figure 6.8. Note that we use the Boolean variable *is_processing* in the event thread to check whether an interrupt has occurred or not.

```
1  bool is_processing = false;
2  ...
3  void* event_thread(void* arg){
4    while (1) {
5      if (nondet_bool()) {
6        is_processing = true;
7        timer_interrupt(); //A hardware interrupt
8        is_processing = false;
9      }
10   }
11   pthread_exit(NULL);
12 }
```

Figure 6.8: Event thread to model the hardware interrupt.

Figure 6.9 shows the concurrent execution of the main, monitor and event threads. Here, the main thread starts the monitor and event threads, which then interleave among them in order to monitor the design's progress and watch out for violations of the LTL property (6.2).

6.5 Experimental Evaluation

This section contains the results of applying the continuous verification approach to two case studies: a pulse oximeter equipment and a large embedded software used in a commercial telecommunication product.

Unless stated otherwise, all experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 4 GB of RAM running Linux OS. For all benchmarks, the time

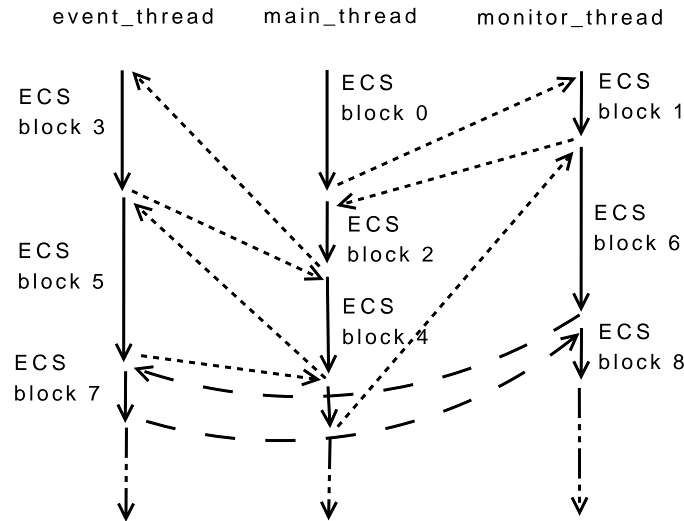

FIGURE 6.9: Concurrent execution of main, monitor and event threads.

limit has been set to 3600 seconds for each individual property. All times given are wall clock time in seconds as measured by the unix *time* command.

All experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 4 GB of RAM running Linux OS. For all benchmarks, the time limit has been set to 3600 seconds for each individual property. All times given are wall clock time in seconds as measured by the unix `time` command through a single execution.

6.5.1 Set-top Box Case Study

In a second case study, we evaluate the feasibility of the elements of the continuous verification approach, i.e., use of the unit tests and function equivalence checking. We use the same NXP set-top box software as in the case study described in Section 3.6, and focus on the *exStbDemo* application, in particular the functions *commandLoop* and *checkCommandParams*.

As described in Section 3.6, the state-of-the-art model checkers fail to verify the functions due to memory limitations and time-outs. However, if we use test cases to guide the state space exploration, we can use the concrete values to drive our symbolic execution engine and then explore the execution paths that were not yet fully explored during dynamic verification. This approach is similar to concolic execution [119, 160], but here we do not generate randomly the concrete values of the test cases with the purpose of maximizing the code coverage. As we can see in Table 6.3, if we use test cases to guide the symbolic execution engine, ESBMC can verify the functions *commandLoop* and *checkCommandParams* with a larger bound, which they both represent the hardest functions to be model checked of the *exStbDemo* application. However, even if we explore only parts of the search space, ESBMC is not yet able to prove or falsify some of the

	Test Case	L	B	P	VC	Time		Properties		
						Solver	Total	Passed	Violated	Fail
1	commandLoop.TC1	545	∞	18	0	<1	4	18	0	0
2	commandLoop.TC2	545	500*	18	3	11	29	18	0	0
3	commandLoop.TC3	545	500*	18	3	11	29	18	0	0
4	commandLoop.TC4	545	17	18	5	8	14	18	0	0
5	commandLoop.TC5	545	∞	18	1	<1	4	18	0	0
6	commandLoop.TC6	545	∞	18	0	<1	4	18	0	0
7	commandLoop.TC7	545	1	18	15	15	19	18	0	0
8	commandLoop.TC8	545	1	18	11	28	31	18	0	0
9	checkCommandParams.TC1	238	17	17	56	<1	9	17	0	0
10	checkCommandParams.TC2	238	17	17	36	<1	5	17	0	0
11	checkCommandParams.TC3	238	17	17	37	<1	5	17	0	0
12	checkCommandParams.TC4	238	17	17	36	7	30	17	0	0
13	checkCommandParams.TC5	238	17	17	80	<1	50	17	0	0
14	checkCommandParams.TC6	238	17	17	664	15	44	17	0	0
15	checkCommandParams.TC7	238	20*	17	957	37	78	17	0	0
16	checkCommandParams.TC8	238	20*	17	1117	170	215	17	0	0

TABLE 6.3: Results for running the test cases for the functions *commandLoop* and *checkCommandParams*.

properties in the functions *commandLoop* (see lines 2 and 3) and *checkCommandParams* (see lines 15 and 16) due to unwinding violations. In any case, the test cases are still useful since the verification of the functions are not completely deterministic, i.e., we still have verification conditions to be checked by the SMT back-end, as we can see in the column VC of the Table 6.3, which shows the total number of generated verification conditions. However, the generalization of the test cases does not produce significant results here since we only have a small number of test cases available and ESBMC thus still runs out of memory or time out to check the properties of the functions *commandLoop* and *checkCommandParams*.

	Function	L	B	P	Time		Properties			Product Releases			
					Solver	Total	Passed	Violated	Fail	PR10	PR11	PR12	PR13
1	threadRename	6	17	0	<1	3	0	0	0	X			
2	fileExists	19	17	0	<1	3	0	0	0	X			
3	readLine	27	17	11	<1	3	1	0	0	X			
4	getCommand	269	17	61	<1	3	61	0	0	X	N/3		N/3
5	powerDown	9	17	0	<1	2	0	0	0	X			
6	digitStart	12	17	0	<1	2	0	0	0	X	Y/2		
7	digitAdd	34	17	2	<1	2	2	0	0	X	Y/2		
8	checkEndOfPvrStream	32	17	13	<1	2	13	0	0	X			Y/2
9	checkEndOfMediaStream	28	17	1	<1	2	1	0	0	X			
10	commandLoop	545	17	53	M_f	M_f	-	-	-	X	M_f	M_f	
11	checkCommandParams	238	17	269	T_b	T_b	0	0	269	X	T_b	T_b	T_b
12	signal_handler	13	17	0	<1	2	0	0	0	X			
13	setupFBResolution	29	17	0	<1	2	0	0	0	X	Y/3	Y/3	Y/2
14	setupFramebuffers	115	17	8	<1	3	8	0	0	X	N/3	N/2	N/2
15	main_Thread	68	17	4	<1	4	4	0	0	X		Y/3	Y/3
16	set_to_raw	8	17	0	<1	3	0	0	0	X			
17	set_to_buffered	8	17	0	<1	2	0	0	0	X		N/2	

TABLE 6.4: Results for checking the equivalence between the functions of the *exStbDemo* application.

We also had access, from the NXP development team, to four different product releases (PRs) that contain the application *exStbDemo*. Based on these four PRs, we identified the functions and methods that have actually been modified and focus on these since the computational effort to re-verify each system build from scratch, in principle, is too high. The four PRs are shown on the right-hand side in Table 6.4 as PR 10, 11, 12, 13. The development time of each PR is about one month and each one contains new features, enhancements (through *refactoring*), and bug fixes. We use PR10 as a reference (and starting point) to compare with PR11, PR11 to compare against PR12 and so on.

Similar to Matsumoto et al. [121], in order to check the equivalence between the functions, we first check the difference between them using the command *diff* in the "old" and "new" releases (i.e., using the command diff *file1.c file2.c* `-p`) to identify the textual differences and then check whether the modified functions are equivalent using our SMT-based bounded model checking as described in Section 6.2. The functions in lines 1 and 5-14 shown in Table 6.4 do not present input/output relations, but they modify global variables; and we are thus able to check the equivalence of them. The notation $N/3$ in line 4 and column PR11 of Table 6.4 means that the function *getCommand* is not equivalent and it takes about 4 seconds to be checked. The notation $Y/2$ in line 6 and column PR11 of Table 6.4 means that the function *digitStart* is different using the command *diff*, but it is functionally equivalent using our SMT-based bounded model checking procedure and it takes 2 seconds to be checked. Blank entries indicate that the respective function is unchanged compared to the previous PR.

As we can see in Table 6.4, each PR only changes a few functions, but while 7 functions remain unchanged over all PRs, there are changes in each individual PR. In particular, the function *getCommand* is not equivalent in PR10, PR11 and PR13; the function *setupFramebuffers* is not equivalent over all product releases; while the function *set_to_buffered* is not equivalent in PR10 and PR12. In summary, we have 19 changes over all PRs, where 8 changes are proved to be equivalent, 6 changes are proved to be not equivalent and we fail to check 5 changes (note that we are not able to prove the equivalence of the functions *commandLoop* and *checkCommandParams* due to memory and time limitations respectively).

From this set of experiments, we conclude that the continuous verification approach can potentially reduce the verification time since the functions 4, 14 and 17 are modified, but they remain functionally equivalent, function 10 (which represents one of the hardest functions) is not modified in PR13, and functions 1-3, 9, 12 and 16 are not modified at all. However, as the verification times of these functions (except for function 10) are small, the advantage of the continuous verification approach is not as substantial as expected. However, if there would be more functional correctness assertions in the code, re-verification would be more expensive and the continuous verification approach would be more advantageous.

6.5.2 Medical Device Case Study

In order to check ESBMC's performance in verifying temporal properties, we analyzed the embedded software of a pulse oximeter device, which is composed of device drivers (i.e., display, keyboard, serial, sensor, and timer) that are 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 embedded software has approximately 3500 lines of ANSI-C code and 80 functions.

In order to meet the application's deadline, there are 100 lines of Assembly code that are responsible for writing text messages to the LCD hardware. ESBMC does not verify Assembly code and as a result we execute this part of the code dynamically only by writing diagnostic messages to a buffer so that we are able to examine the call stack (each message written to the buffer reports the source file, line number, severity, and diagnostic text). These diagnostic messages have been proved to be quite useful to evaluate flight software systems and aid test engineer to understand the system behaviour [82].

Table 6.5 summarizes the results in the usual format. The column *Property* gives the identifier of the LTL property that has been checked. The column L gives the number of lines of code of the test program while the column T reports the total number of threads. Note that T is always three because here we only have the *main*, *monitor* and *event* threads that are running, as described in Section 6.4. The column B provides the unwinding bound for each loop while C is the context switch bound. We use the symbol - to denote that C has not been specified, i.e., we do not restrict the context switch bound. The *Time* column provides the time in seconds while the column #FI/#I provides the total number of failed and generated interleavings respectively. The superscript † means that we injected a fault in the module.

We checked two types of LTL properties (i.e., AG(p → F r) and AG(p)) over different modules of the pulse oximeter. Here, we describe in detail each property presented in Table 6.5:

P1: Whenever the start button is pressed, the application will eventually be initialized (i.e., AG (startButton → F startApp). To check this property, we included two additional Boolean variables into the program *menu_app* to indicate whether the start button has been pressed (represented by *startButton*) and whether the application has been initialized (represented by *startApp*).

P2: It is possible to get to a state where the next position of the buffer is less than its total size (i.e., AG (next < buffer_size)). To check this property, we did not change the program *log* since *next* and *buffer_size* are already declared as global variables.

	Test program	Property	L	T	B	C	Time	#FI / #I
1	menu_app	P1	847	3	2	-	16	0/3003
			847	3	3	20	271	0/50456
			847	3	4	20	625	0/87386
2	menu_app†	P1	847	3	2	-	9	663/3003
			847	3	3	20	121	7584/50456
			847	3	4	20	218	12548/87386
3	log	P2	135	3	2	-	12	0/12
			135	3	3	-	820	0/22
			135	3	4	10	1149	0/8
4	log†	P2	135	3	2	-	1	12/16
			135	3	3	-	3	27/31
			135	3	4	-	5	48/52
5	keyboard	P3	49	3	2	-	7	0/120
			49	3	3	-	80	0/1001
			49	3	4	-	1007	0/8568
6	keyboard†	P3	49	3	2	-	1	2/6
			49	3	3	-	1	3/8
			49	3	4	-	1	4/10
7	serial	P4	165	3	2	-	16	0/1287
			165	3	3	-	980	0/50388
			165	3	4	10	21	0/1023
8	serial†	P4	165	3	2	-	3	347/1287
			165	3	3	-	147	17286/50388
			165	3	4	10	3	189/1023
9	sensor	P5	584	3	2	20	333	0/27768
			584	3	3	20	1452	0/54900
			584	3	4	10	12	0/330
10	sensor†	P5	584	3	2	20	56	4420/18096
			584	3	3	20	211	4655/26326
			584	3	4	20	365	4655/26708

Table 6.5: Results of the LTL properties verification of the pulse oximeter.

P3: Whenever the bit 0 of the micro-controller port is set to high, the start button of the pulse oximeter keyboard will eventually be detected (i.e., AG (BIT0 $\rightarrow$ F startButton)). To check this property, we included two additional Boolean variables into the program *keyboard* to indicate whether the first bit of the micro-controller port has been set to high (represented by *BIT0*) and whether the start button has been detected (represented by *startButton*).

P4: Whenever we set the baud rate of the micro-controller serial port to 1200 bits/second, then its serial register will eventually be configured (i.e., AG(br1200 $\rightarrow$ F reg1200)). To check this property, we included two additional Boolean variables into the program *serial* to indicate whether the baud rate has been set to 1200 bits/second (represented by *br1200*) and whether the serial register has been con-

figured (represented by *reg1200*).

P5: Whenever we receive the synchronism bit from the pulse oximeter sensor, its content will eventually be stored into the checksum[2] array (i.e., AG(sync_byte → F checksum_stored)). To check this property, we included two additional Boolean variables into the program *sensor* to indicate whether the synchronization byte has been received (represented by *sync_byte*) and whether it has been stored into the checksum array (represented by *checksum_stored*).

Note that we have to manually introduce additional Boolean variables into all test programs (except for the test program *log*) in order to indicate whether a given event has occurred or not. Note further that we have to manually merge the resulting C-monitor into the code as we described in Section 6.4.

As shown in Table 6.5, we also injected faults in all test programs as follows:

menu_app: We do not initialize the application after the start button is pressed.

log: We change the program statements so that in a situation where the *next* index is at the end of the array *buffer*, an overflowing index by one byte can occur, i.e., we replace the program statement

$$next = (next + 1)\%buffer_size;$$

by

$$next\% = buffer_size;$$

$$next+ = 1;$$

keyboard: We comment out the *break* statement (of the following program statement that is included into a switch-case: *case START: command=startButton; break;*) so that if *START* was pressed, the code would fall through to the next line, and have the wrong value assigned to *startButton*.

serial: Similar to the faulty *keyboard* program, we comment out the *break* statement that selects the baud rate so that the case statement selecting the baud rate would, in the case of 1200 baud, fall through a case and set the timer to a wrong value (i.e., 2400).

sensor: We replaced assignments to an internal flag (that detects the synchronization bit) by non-deterministic values (i.e., $flag = nondet_bool()\ ?\ true\ :\ false$).

[2]The checksum of the pulse oximeter detects errors that might be introduced during the data collection.

The pulse oximeter software is a reactive system and does not terminate. In general, ESBMC can thus only check the LTL properties up to a certain unwinding and context-switch bounds as shown in Table 6.5. However, for smaller values of the unwinding bound B, the number of context switches is limited, and ESBMC is able to model check the properties without a specified upper bound on the context switches (denoted by - in Table 6.5). Additionally, if a given LTL property does not hold in the test program, ESBMC is able to detect the violation in few seconds and about 15% of the generated interleavings actually fail.

6.6 Related Work

One way of tackling large verification problems is to leverage both parallelism and search diversity [93]. Holzmann et al. describe the Swarm tool that allows using different search strategies on multi-core machines [93]. It is the main interface to the SPIN model checker to verify larger systems. This approach, however, involves large communication overhead and does not take into account information from the software configuration management (SCM) system in order to focus the verification effort on new and/or modified functions. In another related work, Holzmann et al. explore the availability of large chunks of memory in order to explore more exhaustively the state space. However, the authors do not consider that the search modes implemented in the SPIN model checker (mainly based on depth-first search) still remain the main performance bottleneck to verify larger system [92].

Peled proposes a set of combinations between model checking and testing, which includes black box checking, adaptive model checking, and unit checking [147]. However, he does not consider the development history from the SCM system and also uses explicit model checking based on automata theory, which does not scale well due to the number of program variables and data type widths [64]. In addition, Peled only describes the techniques, but does not apply it to any commercial product. Gunter and Peled [85] extend this approach by proposing a symbolic verification approach for a unit of code, also called unit checking. The authors, however, apply this approach only to check whether a complex number diverges to infinity, while we focus on the verification of large embedded software.

Sen et al. propose an approach called *concolic testing* that aims to simultaneously execute a program concretely and symbolically by combining random testing with symbolic execution [77, 119, 160]. It thus removes partially the limitations of random testing (i.e., coverage) and symbolic execution (i.e., scalability). This approach, however, can fail to compute concrete values that satisfy a given (large) path constraint (which can involve complex expressions) due to the solver performance. However, in [119], Majumdar and Sen proposed an approach called *hybrid concolic testing* that combines random and

concolic testing and scale it to large software implementations (e.g., for programs with up to 150K lines of code).

Godlin and Strichman describe an approach called *regression verification* that aims to prove the equivalence of two C programs [78, 167]. Their approach is built on top of CBMC [42], which thus eliminates loops and recursive functions and can handle almost all of the features of ANSI-C. In order to make their approach to scale, the authors isolate functions from their callees and abstract them with uninterpreted functions. Godlin and Strichman apply their approach to random and industrial programs (e.g., from 300 to 3000 lines of code); and they are thus able to check equivalence in minutes. Matsumoto et al. also describe an approach to check the equivalence of two C programs using the SMT solver CVC [121]. Before checking the equivalence of the two programs, the authors first identify the textual differences between them in order to get hints where the equivalence must be checked. However, in their approach the authors restrict the C programs to be checked (e.g., no pointer uses) due to limitations of their symbolic execution engine.

6.7 Conclusions

For large embedded software, SMT-Based bounded model checking suffers from the state space explosion problem. In this chapter we proposed an approach called *continuous verification* to detect design errors as quickly as possible by looking at the software configuration management system and by combining dynamic and static verification to reduce the state space to be explored. As a result, the continuous verification approach and the combination of different encodings and solvers allowed us to explore more exhaustively the state space of the program. Controlled experiments using a case study from the telecommunications domain with more than 10K of lines of C code shows that this approach can potentially improve the error-detection capability and reduce the verification time. However, the advantage of the continuous verification approach is not as substantial as expected, and in this sense we achieved the fourth objective stated in Section 1.2 only partially. If there would be more functional correctness assertions in our case study, re-verification would be more expensive and the continuous verification approach would then be more advantageous.

Chapter 7

Conclusions

In this thesis, we investigated SMT-based verification for single- and multi-threaded ANSI-C programs, focusing in particular on embedded software. As a first step, we described a new set of encodings that allow us to reason accurately about bit operations, unions, fixed-point arithmetic, arrays, pointers (and pointer arithmetic) and dynamic memory allocation and implemented it in the ESBMC tool. We integrated the SMT solvers CVC3, Boolector, and Z3 into ESBMC and evaluated them using both standard software model checking benchmarks and typical embedded software applications from telecommunications, control systems, and medical devices. Our experiments constitute, to the best of our knowledge, the first substantial evaluation of SMT-based bounded model checking on industrial applications. The results show that our approach outperforms CBMC [42] and SMT-CBMC [11] if we consider the verification of embedded software and thus confirm that we successfully met the first objective stated in Section 1.2. ESBMC is able to model check ANSI-C programs that involve tight interplay between non-linear arithmetic, bit operations, pointers and array manipulations. In addition, it was able to find undiscovered bugs in the NECLA, PowerStone, Siemens, SNU-RT, VERISEC and WCET benchmarks related to arithmetic overflow, buffer overflow, invalid pointers and pointer arithmetic.

Compared to ESBMC, SMT-CBMC still has limitations not only in the verification time (due to the lack of simplification based on high-level information), but also in the encodings of important ANSI-C constructs used in embedded software. CBMC is a SAT-based BMC tool for full ANSI-C, but it has limitations due to the fact that the size of the propositional formulae increases significantly in the presence of large datapaths and high-level information is lost when the verification conditions are converted into propositional logic (preventing potential optimizations to reduce the state space to be explored). Its prototype SMT-based back-end is still unstable and fails on a large fraction of our benchmarks. We also improved considerably the performance of SMT-based bounded model checking for embedded software by making use of high-level information to simplify the unwound formula and by determining the best representation

(i.e., SMT logics) to model the program variables and thus successfully met the third objective stated in Section 1.2. As a result, our approach represents a promising direction to improve the state space coverage and to verify quickly properties in larger state spaces using bounded model checking.

Despite the large body of (theoretical) research in the verification of multi-threaded systems, there are only few formal verification tools that analyze multi-threaded programs with shared variables and locks. As a second step, we presented the lazy, schedule recording, and underapproximation and widening algorithms to model check multi-threaded ANSI-C software with shared variable, mutexes and conditions. In the lazy approach, we generate all possible interleavings and call the BMC procedure on each of them individually, until we either find a bug, or have systematically explored all interleavings. In the schedule recording approach, we encode all possible interleavings into one single formula and then exploit the high speed of the SMT solvers. In the underapproximation-widening approach, we reduce the state space by abstracting the number of state variables and interleavings from the proofs of unsatisfiability generated by the SMT solvers. In all three approaches, we bound the number of context-switches and use partial-order reduction techniques to reduce the number of interleavings explored. We also presented our modelling of the synchronization primitives of the Pthread library that allowed us to detect not only atomicity and order violations, but also local and global deadlock, that previous attempts are unable to find [73, 102, 103, 152]. Surprisingly, our approach to check constraints lazily is extremely fast for programs that contain errors and to a lesser extent even for safe programs in which the number of threads and context switches grows quickly. Our experimental results also show that the lazy approach generally outperforms not only the schedule recording and underapproximation and widening approaches, but also the CHESS [136] and SATABS [44] tools on several non-trivial benchmarks as well as state-of-the-art techniques that combine classic partial order reduction methods with symbolic algorithms. With these approaches to verify multi-threaded software (with shared variables) we successfully met the second objective stated in Section 1.2.

For large embedded software, SMT-based bounded model checking still suffers from the state space explosion problem. Finally, as a third step, we defined and evaluated the *continuous verification* approach, which combines existing ideas of software engineering (e.g., continuous integration [70]) and formal verification (e.g., equivalence checking [30]) communities. We applied the elements of the continuous verification approach to the verification of small and large embedded software used in the medical and telecommunications domains. In the medical device case study, ESBMC can only check the LTL properties up to a certain unwinding and context-switch bounds since the software is a reactive system and does not terminate. However, if a given LTL property does not hold, ESBMC is able to detect the violation in few seconds and about 15% of the generated interleavings actually fail. In the telecommunication case study, we concluded that the continuous verification approach can potentially reduce the verification time of large

embedded software systems. However, as the complete verification time of the functions under observation is small, the advantage of the continuous verification approach is not so pronounced and in this sense we achieved the fourth objective stated in Section 1.2 only partially; if there would be more (functional correctness) assertions in the code, re-verification would be more expensive and the continuous verification approach would then be more advantageous.

7.1 Main Contributions

Our work makes two major contributions. First, we describe the details of an accurate translation from single-threaded ANSI-C programs into quantifier-free formulae using the logics QF_AUFBV and QF_AUFLIRA from the SMT-LIB [164]. We demonstrate that our encoding and optimizations improve the performance of software model checking for a wide range of software systems, with a particular emphasis on embedded software, if compared to other approaches proposed by Kroening [105] and Armando et al. [11]. To the best of our knowledge, no SMT-based BMC tool existed that can reliably handle full ANSI-C. Additionally, we show that our encoding allows us to reason about arithmetic overflow and to verify programs that make use of bit-level, pointers, unions and fixed-point arithmetic, where previous attempts fail [105, 11, 71, 88]. We also use three different SMT solvers (Boolector, CVC3, and Z3) in order to check the effectiveness of our encoding techniques. This evaluation thus allows us to quantitatively assess the benefit of using SMT solvers in software verification; in addition, it also provides direction for the new development of SMT solvers.

The second main contribution is in the combination of symbolic model checking with explicit state space exploration that underlies our lazy, schedule recording and underapproximation-widening approaches to handling multi-threaded software. In particular, the difference between our approach and that of Cimatti et al. [39] is that we use BMC instead of predicate abstraction and we implement a realistic scheduler, i.e., our scheduler may preempt a thread at any visible instruction in its execution, whereas Cimatti et al. [39] encodes the semantics of the non-preempting SystemC scheduler. To the best of our knowledge, the lazy approach has not been described or evaluated in the literature. Similarly, the underapproximation-widening approach has not been used for bounded model checking of multi-threaded software; also our approach uses a different encoding based on the notion of effective context-switch blocks. The difference between our schedule recording and Gupta et al. [103] is that they work in a fully symbolic context.

7.2 Future Work Directions

Conceptually, software debugging can be divided into three main steps: *fault detection*, *fault localization* and *fault correction*. In order to detect faults in multi-threaded software, all possible thread interleavings must be systematically explored, which is particularly difficult for traditional testing. Fault localization (and thus correction) is in general a very time-consuming process in software development, which becomes even worse for multi-threaded software mainly due to the non-determinism of the thread interleavings. A number of different approaches have been proposed in the literature to localize faults in software systems, including, for example, slicing, mutation testing, trace-based analysis, delta-debugging, model-based debugging and model checking (for a recent survey we refer the reader to [122]). Apart from these approaches, the debugging time can be substantially reduced if an automatic method is used to localize faults in multi-threaded software. As future work, we thus intend to develop a new method for fault localization in multi-threaded C programs using model checking. In particular, we intend to extend the sequential fault localization method proposed by Griesmayer et al. [81] to localise faults in multi-threaded programs and thus evaluate this approach with industrial benchmarks using our ESBMC model checker.

We also intend to investigate the application of Craig interpolation [125] and the lazy abstraction paradigm [127] to the verification of multi-threaded software. However, differently from [127], we intend to use Craig interpolation to derive thread invariants and not just for unfolding sequential programs. The interpolation-based model checking algorithm [125] described in Subsection 2.2.3.1 requires an unfolding of the entire program up to some bound k. In contrast to [125], we would like to use a lazy abstraction method (similar to [127]) so that we can apply the SMT solvers to individual program paths in order to reduce the burden on the solver. In order to achieve this goal, we would have to refine the model using interpolants derived from refuting program paths. This would avoid the high cost of computing the predicate image operator (as described in Subsection 2.2.3.1), allowing us to improve substantially the performance of the model checker.

Another direction of future work we intend to pursue is to investigate the problem of verifying real-time software using SMT techniques. Most model checkers (e.g., UPPAL [113], TSMV [120] and NuSMV [38]) that reason about timing properties in real-time systems consider that the model is expressed as a timed automata (TA) and they use explicit state-space exploration or BDD-based model checking techniques. To the best of our knowledge, there is only one paper that considers the verification of real-time systems using SMT techniques for checking the satisfiability of the generated formula, which is described by Xu [181]. Xu applies his method to verify liveness and timing properties of the form $F\, m..n\, \phi$ (where m and n represent upper and lower time bounds respectively) in these two models, *Fischer's Protocol* and the *Bridge-crossing problem* [181]. However,

Xu does not support directly real-time software and considers that in the model each transition takes unit time for execution. This assumption is not realistic because for embedded real-time systems, we need mechanisms to assign values to each transition so that these values are the estimated worst-case execution times (WCET) of the respective transition on the selected processor.

7.3 Concluding Remarks

Embedded computer systems are used in a wide range of sophisticated applications, such as mobile phones or set-top boxes providing internet connectivity. The functionality demanded in such applications has increased significantly and an increasing number of functions are implemented in software rather than hardware. Multi-core processors with scalable shared memory have thus become popular in embedded systems. In turn, the verification of the software design and the correctness of its multi-threaded implementations has become increasingly difficult. This thesis, in particular, proposed a comprehensive and implemented SMT-based bounded model checking procedure to reason accurately and effectively about single- and multi-threaded software in embedded systems by exploiting SMT solvers in order to prune the property and data dependent search space and to remove interleavings that are not relevant by analyzing the proof of unsatisfiability. However, the development of reliable embedded software is a complex problem [100] and software verification for embedded systems is still in its infancy since it has been little explored by the research community. Tools for model checking software are still under heavy development, as observed recently by [81]. Therefore, the development of software model checkers based on SMT techniques is still a fertile research area that should be further explored.

Appendix A

ESBMC plug-in

This appendix describes the Eclipse plug-in for ESBMC in order to assist the verification engineer when using the ESBMC model checker. The main ESBMC plug-in window consists of five tabs that allow you to set the different run-time options of the ESBMC model checker. This plug-in was developed with the help of Qiang Li during his summer internship.

The ESBMC plug-in is developed in Eclipse Helios [87], release 3.6 with the Java Run-Time Environment (JRE) 1.6 running on a Linux operating system. This appendix describes only the main features of the ESBMC plug-in. For further information (e.g., how to install, how to use, and how to uninstall the ESBMC plug-in), we refer the reader to the user manual of the ESBMC plug-in available on-line at [52].

A.1 Front-end Options

Figure A.1 shows the options available in the ESBMC front-end, which are described as follows:

- **Use current file:** You can analyze the file that is open in your current editor. If you want to analyze other files located in your file system, then uncheck the box Use current file and click on Browse to choose the program that you want to analyze. If your application consists of more than a single C program, then you can specify them as a sequence (e.g., /home/esbmc/file1.c file2.c).
- **Set include path:** You can set the include path, which contains the .h files, by clicking on the Browse button.
- **Define preprocessor macro:** You can define C preprocessor macro in this text area by just providing the name without # as directives.

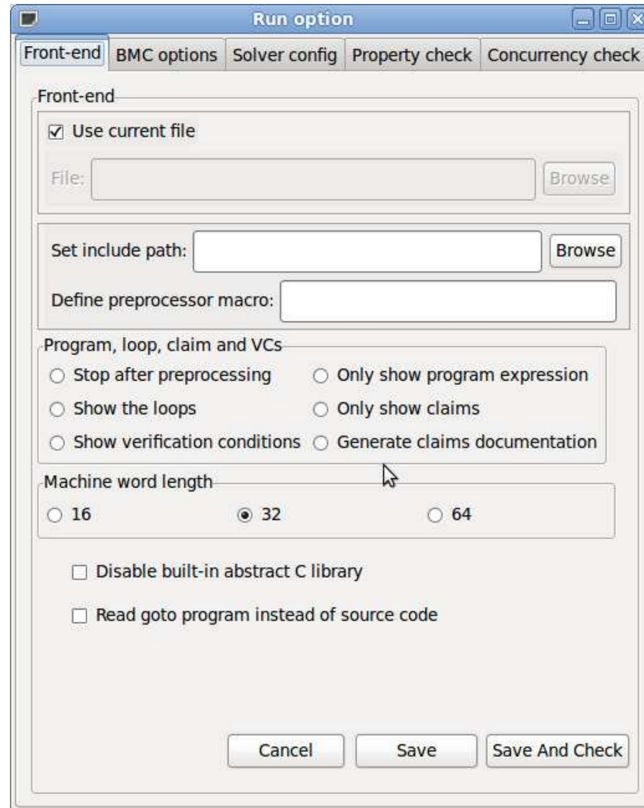

Figure A.1: Front-end options.

- **Program, loop, claim and VCs:** You can choose the options to show the preprocessed program, all the claims (or properties) given as assertions by the designers as well as a range of language-specific safety (such as the absence of arithmetic under- and overflow, out-of-bounds array indexing, or nil-pointer dereferencing), show the verifications conditions that are generated during BMC, the identification of the loops in the program, the expressions of the program in single static assignment (SSA) form, and the documentation (in Latex) of the generated claims. However, note that all these options are mutually exclusive, because they produce an output to the same (Console) view, i.e., you can visualize one of them on each time.

- **Machine word length:** You can set your machine word length, the default is 32.

- **Disable built-in abstract C library:** The C programs usually use functions of the ANSI-C library (e.g., strcmp, printf), which contain information that are irrelevant from the verification point of view. We thus provide an abstract ANSI-C library implemented internally in the model checker, which comprises a small set of the functions. If you do not want to use the built-in ANSI-C library, then you should select this option.

- **Read goto program instead of source code:** This option allows you to model check the goto programs (i.e., control-flow graphs) generated by the goto-cc

tool [179].

A.2 BMC Options

The BMC options of the ESBMC plug-in are shown in Figure A.2. It consists of the following options:

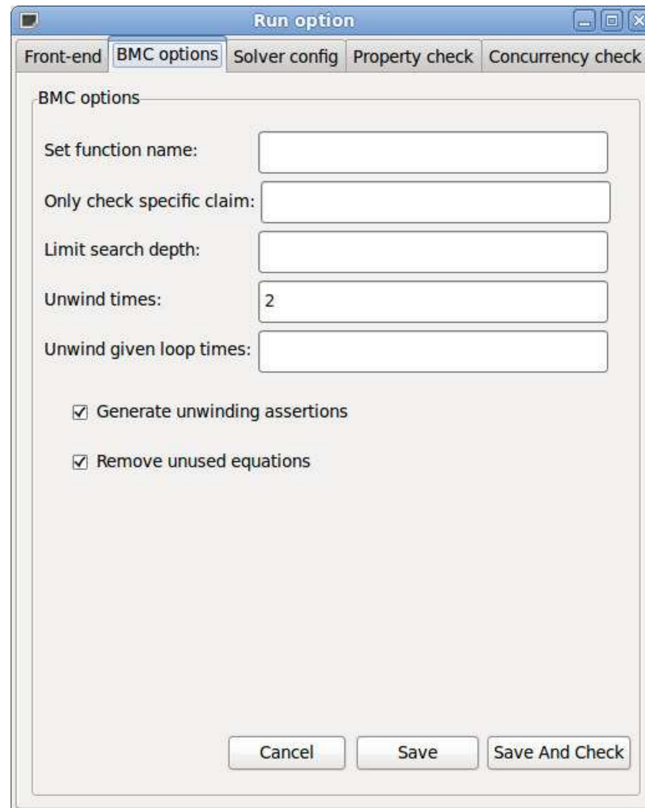

FIGURE A.2: BMC options.

- **Set function name:** You can set the main function name here.
- **Only check specific claim:** You can check for a specific claim, so please input the number of the identification of the claim.
- **Limit search depth:** You can limit search depth, so please input an integer number.
- **Unwind times:** Set the unwind bound in here, the default is 2. You have to provide an integer number.
- **Unwind given loop times:** This option allows you to unwind a specific loop in your program. Here, you should provide the identification of the loop.

- **Do not generate unwinding assertions:** If you do not want to check that you have unrolled enough the loops in your program, then you should select this option.
- **Do not remove unused equations:** If it is unchecked, unused equations are removed automatically during the symbolic execution.

A.3 SMT Solver Configuration

The solver configuration options tab is shown in Figure A.3. It consists of the following options:

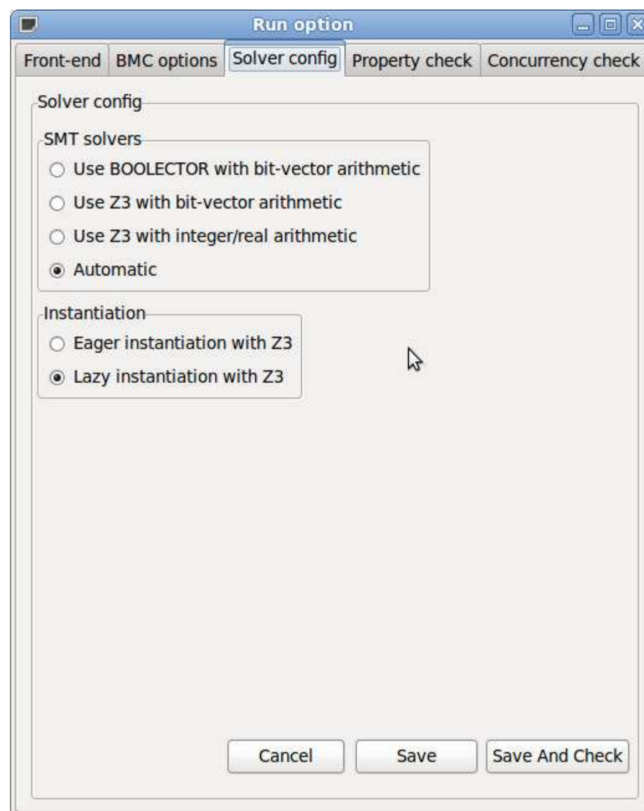

FIGURE A.3: SMT Solver Configuration.

- **SMT solvers:** If you choose the first one, the model checker will use BOOLECTOR with bit-vector arithmetic as a decision procedure to model check your program. The second option is to use Z3 with bit-vector arithmetic, and the third ESBMC is Z3 with integer/real arithmetic. The last option, which is the default option, will determine the best solver and encoding to be used according to the verification conditions that are generated from your C program.
- **Instantiation:** You can choose either eager or lazy instantiation to solve the SMT instances with Z3 (lazy is the default option).

A.4 Property Check

You can select which safety properties you want to check in your single-threaded program as show in Figure A.4. This tab consists of the following options:

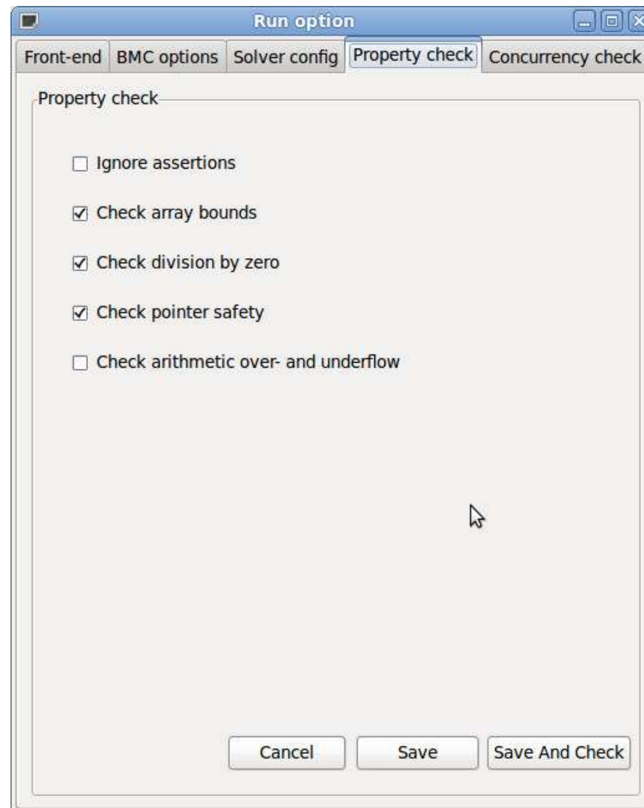

FIGURE A.4: Property check.

- **Ignore assertions:** This option ignores all assertions in your C program.
- **Do not do array bounds check:** This option does not allow ESBMC to generate verification conditions related to checking out-of-bounds array indexing.
- **Do not do division by zero check:** This option does not allow ESBMC to generate verification conditions related to checking division by zero in arithmetic expressions.
- **Do not do pointer check:** This option does not allow ESBMC to generate verification conditions related to checking nil-pointer dereferencing.
- **Enable arithmetic over- and underflow check:** This option does not allow ESBMC to generate verification conditions related to checking arithmetic over- and underflow.

A.5 Concurrency Check

You can select which approaches and properties you want in order to verify in your multi-threaded program as shown in Figure A.5.

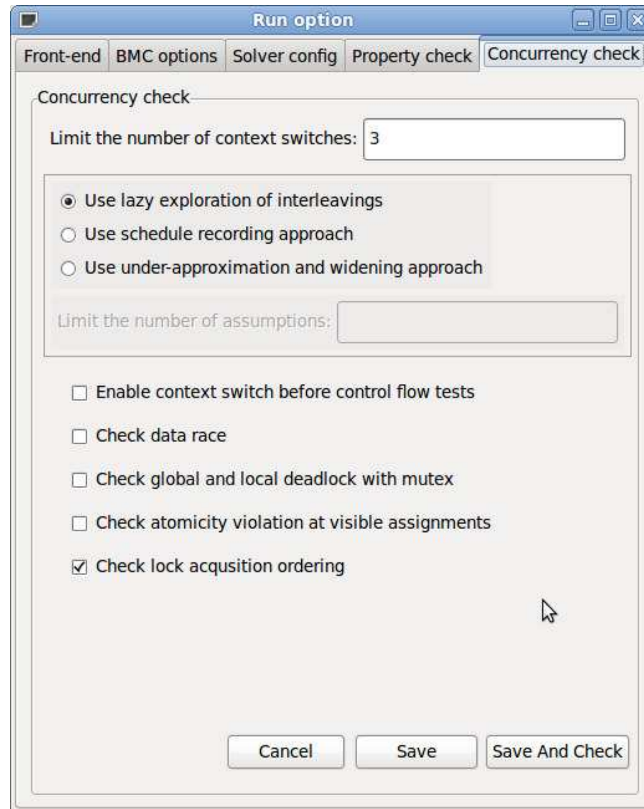

FIGURE A.5: Concurrency check.

- Limit the number of context switches: Limit the number of context switches allowed per each thread. You have to provide an integer number here.
- Use schedule recording approach: This option allows ESBMC to encode all possible interleavings into one single formula and then exploit the high speed of the SMT solvers.
- Use under-approximation and winding approach: This option allows ESBMC to check models with an increasing set of allowed interleavings.
- Limit the number of assumptions: If you choose Use under-approximation and winding approach, then you can limit the number of assumptions in the UW approach. You have to provide an integer number in the text area.
- Enable global and local deadlock check with mutex: This option checks whether all threads wait for a mutex (global deadlock) or whether some of the threads form a waiting cycle (local deadlock).

- Enable data races check: This option checks whether multiple threads perform unsynchronized accesses to shared data.
- Do not do lock acquisition ordering check: This option checks for unintended sequence of lock and unlock operations among the threads.
- Enable atomicity violation check at visible assignments: This option allows ESBMC to break visible statements to check if a region of code executes atomically.
- Enable context switch before control flow tests: This option allows ESBMC to simulate the effect of a context switch right after a visible test by hoisting the test out of the conditional, and assigning its result to a new auxiliary variable.

A.6 Counterexample, Property Violation, and Claim Views

In order to model check your C program, you should click on the *Save and Check* button as shown in Figure A.1, or click on *Verify Current File* menu item or simply type the shortcut *CTRL+ALT+C*. When the verification fails (i.e., the property does not hold in the program), you can see details of the property violation and counterexample (or trace to reproduce the violation) in the corresponding views as shown in the bottom of Figure A.6.

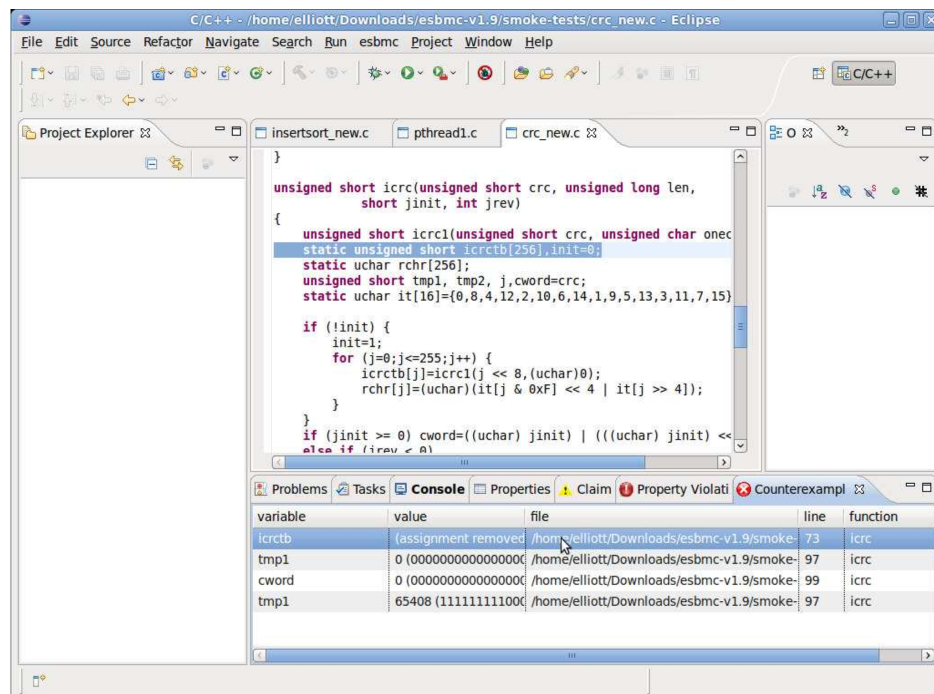

FIGURE A.6: Counterexample view.

If you double click on the variable name in the counterexample view, then you go directly to the corresponding line in the program where the error is located. The property

violation and claim views work in the same way as in the counterexample view, i.e., you should click in one line of the table in order to go directly to the corresponding line in the program. If you need to obtain more information about the results of other options of the ESBMC model checker (e.g., show program only, show loops), then you can easily visualize them in the Eclipse console.

Appendix B

Static Analysis Benchmarks

This appendix expands the experimental results that we presented in Table 3.3 of Chapter 3 by providing details of the verification time for the considered static analysis benchmarks.

All experiments were conducted on an otherwise idle Intel Xeon 5160, 3GHz server with 4 GB of RAM running Linux OS. For all benchmarks, the time limit has been set to 3600 seconds for each individual property. All times given are wall clock time in seconds as measured by the unix time command through a single execution.

We invoked ESBMC by setting the file name, the unwinding bound and enabling the arithmetic under- and overflow check as well as string abstraction (i.e., `esbmc file --unwind` B `--overflow-check --string-abstraction`).

B.1 EUREKA Suite

Table B.1 shows the results of applying ESBMC to the verification of the programs from the EUREKA suite. Note that the EUREKA suite only contains correct programs and ESBMC is able to verify all properties.

B.2 POWERSTONE Suite

Table B.2 shows the results of applying ESBMC to the verification of the programs from the PowerStone suite. Note that that the lines that are marked as bold indicate undiscovered bugs that ESBMC was able to find.

	Module	*L*	*B*	*P*	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	EUREKA_bf20	49	21	41	0.06	1	41	0	0
2	EUREKA_BubbleSort	305	141	160	125.98	335	160	0	0
3	EUREKA_Prim	79	9	41	0.15	1	41	0	0
4	EUREKA_SelectionSort	309	141	156	12.63	155	156	0	0
5	EUREKA_StrCmp	14	1000	6	32	35	6	0	0
6	EUREKA_SumArray	12	1000	7	9	10	7	0	0
7	EUREKA_MinMax	19	1000	9	2	6	9	0	0
-	**Total**	**787**	-	**420**	**181.82**	**543**	**420**	**0**	**0**

Table B.1: Results of applying ESBMC to the verification of the benchmarks from the EUREKA suite.

	Module	*L*	B	*P*	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	POWERSTONE_adpcm	473	55	545	199.35	263	545	0	0
2	POWERSTONE_bcnt	83	17	157	1.14	1	157	0	0
3	***POWERSTONE_blit***	***95***	***1025***	***133***	***11.34***	***17***	***126***	***12***	***0***
4	POWERSTONE_compress	565	120	367	312.29	318	367	0	0
5	POWERSTONE_cr	99	257	22	0.25	8	22	0	0
6	POWERSTONE_engine	291	2	295	0.05	1	295	0	0
7	POWERSTONE_fir	116	34	124	0.36	3	124	0	0
8	POWERSTONE_g3fax	606	2	143	47.69	48	143	0	0
9	POWERSTONE_jpeg	529	5	245	155.3	157	245	0	0
-	**Total**	**2857**	-	**2031**	**728**	**816**	**2019**	**12**	**0**

Table B.2: Results of applying ESBMC to the verification of the benchmarks from the PowerStone suite.

B.3 NECLA Suite

Table B.3 shows the results of applying ESBMC to the verification of the correct programs from the NECLA benchmarks. Note that ESBMC finds three property violations in two programs (*ex13* and *ex28*) from Table B.3, which have been confirmed as true faults by the benchmark creators [97].

Table B.4 shows the results of applying ESBMC to the verification of the bad programs from the NECLA benchmarks. Note that ESBMC was able to verify two programs (*ex25* and *ex40*) from Table B.4 that did not contain any seeded errors; the benchmark creators confirmed that these two programs were misclassified and subsequently changed the error seeding [97].

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	NEC.ex10	72	17	10	0.02	1	10	0	0
2	NEC.ex11	24	1000	3	17.14	30	3	0	0
3	***NEC.ex13***	***9***	***33***	***2***	***0***	***1***	***1***	***1***	***0***
4	NEC.ex14	15	11	5	0	1	5	0	0
5	NEC.ex15	34	2	5	0	1	5	0	0
6	NEC.ex16	34	10000	4	1.47	6	4	0	0
7	NEC.ex17	44	101	14	0.01	1	14	0	0
8	NEC.ex19	28	10	2	0.02	1	2	0	0
9	NEC.ex1	22	513	10	0.27	3	10	0	0
10	NEC.ex21	25	1024	6	0.02	1	6	0	0
11	NEC.ex22	38	51	9	0.01	1	9	0	0
12	NEC.ex23	20	37	1	0.01	1	1	0	0
13	NEC.ex24	78	1000	37	0.11	0.64	37	0	0
14	***NEC.ex28***	***12***	***101***	***11***	***0.01***	***1***	***9***	***2***	***0***
15	NEC.ex29	47	101	32	0.01	1	32	0	0
16	NEC.ex2	39	1025	4	12.21	19	4	0	0
17	NEC.ex30	45	101	16	1.16	3	16	0	0
18	NEC.ex31	13	8	6	0.02	1	6	0	0
19	NEC.ex32	26	1001	4	0.16	1	4	0	0
20	NEC.ex33	35	100	13	0	1	13	0	0
21	NEC.ex34	24	10	7	0.01	1	7	0	0
22	NEC.ex37	26	10	5	0	1	1	0	0
23	NEC.ex38	25	201	16	0	1	16	0	0
24	NEC.ex39	26	100	4	1.1	1	4	0	0
25	NEC.ex42	32	40	12	63.05	87	12	0	0
26	NEC.ex49	15	100	2	0.12	1	2	0	0
27	NEC.ex5	17	100	6	0	1	6	0	0
28	NEC.ex6	20	100	0	1	0	0	0	0
29	NEC.ex7	27	100	3	0.16	1	3	0	0
30	NEC.ex8	19	100	5	0.37	1	5	0	0
-	**Total**	**891**	-	**254**	**98**	**172**	**212**	**3**	**0**

TABLE B.3: Results of applying ESBMC to the verification of the correct benchmarks from the NECLA suite.

B.4 SNU-RT Suite

Table B.5 shows the results of applying ESBMC to the verification of the programs from the SNU-RT suite. Note that ESBMC finds array bounds violations and overflows in arithmetic expressions in four of the SNU-RT benchmarks (*crc_nondet*, *fibcall_nondet*, *insertsort_nondet* and *jfdctint_det*); we confirmed by inspection that these are indeed faults.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	NEC.ex12	23	21	4	0	1	3	1	0
2	NEC.ex20	32	10	12	0.01	1	12	0	0
3	***NEC.ex25***	***26***	***101***	***7***	***0.07***	***2***	***7***	***0***	***0***
4	NEC.ex26	29	101	11	0.09	1	9	2	0
5	NEC.ex27	39	101	9	0.03	1	7	2	0
6	NEC.ex3	25	11	4	0	1	3	1	0
7	***NEC.ex40***	***19***	***101***	***9***	***0.54***	***1***	***9***	***0***	***0***
8	NEC.ex41	22	10	10	32.05	32	6	4	0
9	NEC.ex4	15	1000	6	0.01	1	4	2	0
10	NEC.ex43	112	21	40	4.709	6	27	13	0
-	**Total**	**342**	-	**112**	**37**	**47**	**87**	**25**	**0**

Table B.4: Results of applying ESBMC to the verification of the bad benchmarks from the NECLA suite.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	SNU.bs_det	114	16	11	0.001	1	11	0	0
2	SNU.bs_nondet	120	16	12	0.071	9	12	0	0
3	SNU.crc_det	125	257	18	0.082	8	18	0	0
4	***SNU_crc_nondet***	***126***	***257***	***13***	***0.29***	***7***	***12***	***1***	***0***
5	SNU.fft1_det	218	9	72	0.004	1	72	0	0
6	SNU.fft1k_nondet	158	0	39	0.763	50	39	0	0
7	SNU.fibcall_det	83	10000	2	0.005	1	2	0	0
8	***SNU_fibcall_nondet***	***84***	***10000***	***2***	***0***	***157***	***0***	***2***	***0***
9	SNU.fir_det	314	34	25	0.361	3	25	0	0
10	SNU.fir_nondet	316	34	25	0.326	2	25	0	0
11	SNU.insertsort_det	86	12	17	0.557	1	17	0	0
12	***SNU.insertsort_nondet***	***94***	***12***	***20***	***4.981***	***5***	***14***	***6***	***0***
13	***SNU.jfdctint_det***	***374***	***65***	***331***	***0.471***	***2***	***311***	***20***	***0***
14	SNU.lms_det	258	202	35	4.358	297	35	0	0
15	SNU.lms_nondet	256	202	35	2.488	21	35	0	0
16	SNU.ludcmp_det	144	144	88	0.042	1	88	0	0
17	SNU.matmul_det	81	6	31	0.055	1	31	0	0
18	SNU.qurt_det	164	20	8	0.139	1	8	0	0
19	SNU.select_nondet	117	1	42	0.001	1	42	0	0
20	SNU.sqrt_det	88	20	2	0.002	1	2	0	0
-	**Total**	**3320**	-	**828**	**15**	**570**	**799**	**29**	**0**

Table B.5: Results of applying ESBMC to the verification of the benchmarks from the SNU-RT suite.

B.5 VERISEC Suite

Tables B.6, B.7 and B.8 shows the results of applying ESBMC to the verification of the correct programs from the VERISEC suite. ESBMC finds 15 property violations in nine programs (see programs 67-73, 75 and 76), which have also been confirmed by the benchmark creators [36].

Tables B.9, B.10 and B.11 shows the results of applying ESBMC to the verification of the bad programs from the VERISEC suite.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	VERISEC.ok_apache_full-ok	58	5	42	0.05	1	42	0	0
2	VERISEC.ok_apache_full-ptr-ok	57	5	37	0.05	1	37	0	0
3	VERISEC.ok_apache_simp2-ok	43	5	24	0.03	1	24	0	0
4	VERISEC.ok_apache_simp3-ok	55	5	40	0.04	1	40	0	0
5	VERISEC.ok_apache_strncmp-ok	41	5	25	0.03	1	25	0	0
6	VERISEC.ok_bind_expands-vars-ok	89	1	29	0.01	1	29	0	0
7	VERISEC.ok_gxine_simp-ok	31	5	15	0	1	15	0	0
8	VERISEC.ok_libgd_gd-no-entities-ok	117	3	30	0.08	1	30	0	0
9	VERISEC.ok_libgd_gd-simp-ok	95	3	28	0.05	1	28	0	0
10	VERISEC.ok_MADWiFi_no-sprintf-ok	53	3	19	0	1	14	0	0
11	VERISEC.ok_NetBSD-libc_anyMeta-int-ok	50	10	12	0.83	2	12	0	0
12	VERISEC.ok_NetBSD-libc_anyMeta-ptr-ok	52	10	11	0.74	2	11	0	0
13	VERISEC.ok_NetBSD-libc_bounds-ok	17	0	2	0	1	2	0	0
14	VERISEC.ok_NetBSD-libc_glob2-int-ok	91	12	28	11.69	19	28	0	0
15	VERISEC.ok_NetBSD-libc_glob2-ptr-ok	92	12	27	21.19	29	27	0	0
16	VERISEC.ok_NetBSD-libc_loop-int-ok	39	4	6	0.01	1	6	0	0
17	VERISEC.ok_NetBSD-libc_loop-ok	24	4	3	0	1	3	0	0
18	VERISEC.ok_NetBSD-libc_loop-ptr-ok	39	4	5	0	1	5	0	0
19	VERISEC.ok_NetBSD-libc_noAnyMeta-int-ok	43	10	10	2.06	3	10	0	0
20	VERISEC.ok_NetBSD-libc_noAnyMeta-ptr-ok	45	10	9	0.63	2	9	0	0
21	VERISEC.ok_OpenSER_cases1-stripFullBoth-arr-inlined-ok	60	10	43	4.04	5	43	0	0
22	VERISEC.ok_OpenSER_cases1-stripFullBoth-arr-ok	59	10	38	3.35	4	38	0	0
23	VERISEC.ok_OpenSER_cases1-stripFullEnd-arr-inlined-ok	54	9	35	0.24	1	35	0	0
24	VERISEC.ok_OpenSER_cases1-stripFullEnd-arr-ok	53	10	30	0.16	1	30	0	0
25	VERISEC.ok_OpenSER_cases1-stripFullStart-arr-inlined-ok	56	10	35	1.77	3	35	0	0
26	VERISEC.ok_OpenSER_cases1-stripFullStart-arr-ok	55	10	30	1.7	2	30	0	0
27	VERISEC.ok_OpenSER_cases1-stripNone-arr-inlined-ok	50	10	27	0.08	1	27	0	0
28	VERISEC.ok_OpenSER_cases1-stripNone-arr-ok	49	10	22	0.05	1	22	0	0
29	VERISEC.ok_OpenSER_cases1-stripSpacesBoth-arr-inlined-ok	56	10	0		1	0	0	0
30	VERISEC.ok_OpenSER_cases1-stripSpacesBoth-arr-ok	55	10	28	1.38	2	28	0	0

TABLE B.6: Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part I.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
31	VERISEC.ok_OpenSER_cases1-stripSpacesEnd-arr-inlined-ok	53	10	30	0.13	1	30	0	0
32	VERISEC.ok_OpenSER_cases1-stripSpacesEnd-arr-ok	52	10	25	0.1	1	25	0	0
33	VERISEC.ok_OpenSER_cases1-stripSpacesStart-arr-inlined-ok	53	10	30	1.16	2	30	0	0
34	VERISEC.ok_OpenSER_cases1-stripSpacesStart-arr-ok	52	10	25	0.93	2	25	0	0
35	VERISEC.ok_OpenSER_cases2-stripFullBoth-arr-inlined-ok	63	10	45	7.54	9	45	0	0
36	VERISEC.ok_OpenSER_cases2-stripFullBoth-arr-ok	62	10	40	6.05	7	40	0	0
37	VERISEC.ok_OpenSER_cases2-stripFullEnd-arr-inlined-ok	57	10	37	0.66	1	37	0	0
38	VERISEC.ok_OpenSER_cases2-stripFullEnd-arr-ok	56	10	32	0.36	1	32	0	0
39	VERISEC.ok_OpenSER_cases2-stripFullStart-arr-inlined-ok	59	10	37	3.25	5	37	0	0
40	VERISEC.ok_OpenSER_cases2-stripFullStart-arr-ok	58	10	32	2.39	3	32	0	0
41	VERISEC.ok_OpenSER_cases2-stripNone-arr-inlined-ok	53	10	29	0.13	1	29	0	0
42	VERISEC.ok_OpenSER_cases2-stripNone-arr-ok	52	10	24	0.07	1	24	0	0
43	VERISEC.ok_OpenSER_cases2-stripSpacesBoth-arr-inlined-ok	59	10	35	4.01	5	35	0	0
44	VERISEC.ok_OpenSER_cases2-stripSpacesBoth-arr-ok	58	10	30	2.7	4	30	0	0
45	VERISEC.ok_OpenSER_cases2-stripSpacesEnd-arr-inlined-ok	56	10	32	0.49	1	32	0	0
46	VERISEC.ok_OpenSER_cases2-stripSpacesEnd-arr-ok	55	10	27	0.18	1	27	0	0
47	VERISEC.ok_OpenSER_cases2-stripSpacesStart-arr-inlined-ok	56	10	32	2.16	3	32	0	0
48	VERISEC.ok_OpenSER_cases2-stripSpacesStart-arr-ok	55	10	27	1.55	2	27	0	0
49	VERISEC.ok_OpenSER_cases3-stripFullBoth-arr-inlined-ok	66	10	47	7.11	8	47	0	0
50	VERISEC.ok_OpenSER_cases3-stripFullBoth-arr-ok	65	10	42	6.52	8	42	0	0
51	VERISEC.ok_OpenSER_cases3-stripFullEnd-arr-inlined-ok	60	10	39	0.75	1	39	0	0
52	VERISEC.ok_OpenSER_cases3-stripFullEnd-arr-ok	59	10	34	0.55	1	34	0	0
53	VERISEC.ok_OpenSER_cases3-stripFullStart-arr-inlined-ok	62	10	39	4.08	5	39	0	0
54	VERISEC.ok_OpenSER_cases3-stripFullStart-arr-ok	61	10	34	3.4	4	34	0	0
55	VERISEC.ok_OpenSER_cases3-stripNone-arr-inlined-ok	56	10	31	0.26	1	31	0	0
56	VERISEC.ok_OpenSER_cases3-stripNone-arr-ok	55	10	26	0.16	1	26	0	0
57	VERISEC.ok_OpenSER_cases3-stripSpacesBoth-arr-inlined-ok	62	10	37	4.32	5	37	0	0
58	VERISEC.ok_OpenSER_cases3-stripSpacesBoth-arr-ok	61	10	32	3.15	4	32	0	0
59	VERISEC.ok_OpenSER_cases3-stripSpacesEnd-arr-inlined-ok	59	10	34	0.53	1	34	0	0
60	VERISEC.ok_OpenSER_cases3-stripSpacesEnd-arr-ok	58	10	29	0.33	1	29	0	0

Table B.7: Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part II.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
61	VERISEC.ok_OpenSER_cases3-stripSpacesStart-arr-inlined-ok	59	10	34	2.15	3	34	0	0
62	VERISEC.ok_OpenSER_cases3-stripSpacesStart-arr-ok	58	10	29	1.79	3	29	0	0
63	VERISEC.ok_samba_simp-ok	22	1	2	0	1	2	0	0
64	VERISEC.ok_sendmail_both-ok	78	5	38	0.42	1	38	0	0
65	VERISEC.ok_sendmail_close-angle-ptr-no-test-ok	44	3	8	0.01	1	8	0	0
66	VERISEC.ok_sendmail_inner-ok	39	4	13	0	1	13	0	0
67	***VERISEC.ok_sendmail_mime7to8-arr-one-char-heavy-test-ok***	***48***	***10***	***15***	***0.29***	***1***	***14***	***1***	***0***
68	***VERISEC.ok_sendmail_mime7to8-arr-one-char-med-test-ok***	***46***	***10***	***15***	***0.17***	***1***	***14***	***1***	***0***
69	***VERISEC.ok_sendmail_mime7to8-arr-one-char-no-test-ok***	***31***	***10***	***7***	***0.01***	***1***	***6***	***1***	***0***
70	***VERISEC.ok_sendmail_mime7to8-arr-three-chars-med-test-ok***	***94***	***10***	***41***	***1.1***	***1***	***38***	***3***	***0***
71	***VERISEC.ok_sendmail_mime7to8-ptr-one-char-heavy-test-ok***	***46***	***10***	***17***	***0.43***	***1***	***16***	***1***	***0***
72	***VERISEC.ok_sendmail_mime7to8-ptr-three-chars-med-test-ok***	***86***	***10***	***45***	***4.36***	***4***	***42***	***3***	***0***
73	***VERISEC.ok_sendmail_mime7to8-ptr-three-chars-no-test-ok***	***48***	***10***	***18***	***0.14***	***1***	***15***	***3***	***0***
74	VERISEC.ok_sendmail_outer-ok	47	4	17	0.02	1	17	0	0
75	***VERISEC.ok_sendmail_prescan-arr-med-test-ok***	***86***	***5***	***21***	***0.06***	***1***	***20***	***1***	***0***
76	***VERISEC.ok_sendmail_prescan-arr-min-test-ok***	***92***	***5***	***21***	***0.05***	***1***	***20***	***1***	***0***
77	VERISEC.ok_sendmail_tTflag-arr-one-loop-ok	23	11	7	0.01	1	7	0	0
78	VERISEC.ok_SpamAssassin_loop-ok	45	7	33	1.94	3	33	0	0
79	VERISEC.wu-ftpd_simple-ok	54	4	18	0	1	18	0	0
80	VERISEC.wu-ftpd_strcpy-strcat-ok	64	5	32	0.01	1	32	0	0
-	**Total**	**4521**	**-**	**2114**	**128**	**211**	**2094**	**15**	**0**

Table B.8: Results of applying ESBMC to the verification of the correct benchmarks from the VERISEC suite - Part III.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
1	VERISEC.apache_full-bad	58	5	39	0.07	1	38	1	0
2	VERISEC.apache_full-ptr-bad	57	5	34	0.08	1	33	1	0
3	VERISEC.apache_simp2-bad	43	5	21	0.03	1	20	1	0
4	VERISEC.apache_simp3-bad	55	5	37	0.07	1	36	1	0
5	VERISEC.apache_strncmp-bad	41	5	22	0.03	1	21	1	0
6	VERISEC.bind_expands-vars-bad	82	1	28	0.01	1	27	1	0
7	VERISEC.cases2_stripSpacesEnd-arr-inlined-bad	53	10	29	0.16	1	26	3	0
8	VERISEC.gxine_simp-bad	31	5	10	0	1	9	1	0
9	VERISEC.libgd_gd-no-entities-bad	120	3	28	0.12	1	25	3	0
10	VERISEC.libgd_gd-simp-bad	98	3	26	0.07	1	23	3	0
11	VERISEC.MADWiFi_no-sprintf-bad	52	3	18	0	1	13	5	0
12	VERISEC.NetBSD-libc_anyMeta-int-bad	50	10	12	0.84	2	7	5	0
13	VERISEC.NetBSD-libc_anyMeta-ptr-bad	52	10	11	0.74	1	6	5	0
14	VERISEC.NetBSD-libc_bounds-bad	17	0	2	0	1	1	1	0
15	VERISEC.NetBSD-libc_glob2-int-bad	91	12	28	19.98	27	16	12	0
16	VERISEC.NetBSD-libc_glob2-ptr-bad	92	12	27	16.48	24	15	12	0
17	VERISEC.NetBSD-libc_loop-bad	24	4	3	0.01	1	2	1	0
18	VERISEC.NetBSD-libc_loop-int-bad	39	4	6	0.01	1	4	2	0
19	VERISEC.NetBSD-libc_loop-ptr-bad	39	4	5	0.01	1	3	2	0
20	VERISEC.NetBSD-libc_noAnyMeta-int-bad	43	10	10	2.53	4	8	2	0
21	VERISEC.NetBSD-libc_noAnyMeta-ptr-bad	45	10	9	0.63	1	5	4	0
22	VERISEC.OpenSER_cases1-stripFullBoth-arr-bad	56	10	35	1.29	2	34	1	0
23	VERISEC.OpenSER_cases1-stripFullBoth-arr-inlined-bad	57	10	40	1.51	3	37	3	0
24	VERISEC.OpenSER_cases1-stripFullEnd-arr-bad	50	10	27	0.12	1	26	1	0
25	VERISEC.OpenSER_cases1-stripFullEnd-arr-inlined-bad	51	9	32	0.16	1	29	3	0
26	VERISEC.OpenSER_cases1-stripFullStart-arr-bad	52	10	27	1.02	2	26	1	0
27	VERISEC.OpenSER_cases1-stripFullStart-arr-inlined-bad	53	10	32	1.25	2	29	3	0
28	VERISEC.OpenSER_cases1-stripNone-arr-bad	46	10	19	0.05	1	18	1	0
29	VERISEC.OpenSER_cases1-stripNone-arr-inlined-bad	47	10	24	0.08	1	21	3	0
30	VERISEC.OpenSER_cases1-stripSpacesBoth-arr-bad	52	10	25	0.87	2	24	1	0

Table B.9: Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part I.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
31	VERISEC.OpenSER_cases1-stripSpacesBoth-arr-inlined-bad	53	10	30	1.13	2	27	3	0
32	VERISEC.OpenSER_cases1-stripSpacesEnd-arr-bad	49	10	22	0.08	1	21	1	0
33	VERISEC.OpenSER_cases1-stripSpacesEnd-arr-inlined-bad	50	10	27	0.12	1	24	3	0
34	VERISEC.OpenSER_cases1-stripSpacesStart-arr-bad	49	10	22	0.76	2	21	1	0
35	VERISEC.OpenSER_cases1-stripSpacesStart-arr-inlined-bad	50	10	27	0.91	2	24	3	0
36	VERISEC.OpenSER_cases2-stripFullBoth-arr-bad	59	10	37	1.72	3	36	1	0
37	VERISEC.OpenSER_cases2-stripFullBoth-arr-inlined-bad	60	10	42	1.55	2	39	3	0
38	VERISEC.OpenSER_cases2-stripFullEnd-arr-bad	53	10	29	0.15	1	28	1	0
39	VERISEC.OpenSER_cases2-stripFullEnd-arr-inlined-bad	54	10	34	0.22	1	31	3	0
40	VERISEC.OpenSER_cases2-stripFullStart-arr-bad	55	10	29	1.42	2	28	1	0
41	VERISEC.OpenSER_cases2-stripFullStart-arr-inlined-bad	56	10	34	1.43	2	31	3	0
42	VERISEC.OpenSER_cases2-stripNone-arr-bad	49	10	21	0.05	1	20	1	0
43	VERISEC.OpenSER_cases2-stripNone-arr-inlined-bad	50	10	26	0.09	1	23	3	0
44	VERISEC.OpenSER_cases2-stripSpacesBoth-arr-bad	55	10	27	0.99	2	26	1	0
45	VERISEC.OpenSER_cases2-stripSpacesBoth-arr-inlined-bad	56	10	32	1.11	2	29	3	0
46	VERISEC.OpenSER_cases2-stripSpacesEnd-arr-bad	52	10	24	0.11	1	23	1	0
47	VERISEC.OpenSER_cases2-stripSpacesEnd-arr-inlined-bad	54	10	29	0.16	1	26	3	0
48	VERISEC.OpenSER_cases2-stripSpacesStart-arr-bad	52	10	24	0.79	1	23	1	0
49	VERISEC.OpenSER_cases2-stripSpacesStart-arr-inlined-bad	53	10	29	1.02	2	26	3	0
50	VERISEC.OpenSER_cases3-stripFullBoth-arr-bad	62	10	39	1.9	3	38	1	0
51	VERISEC.OpenSER_cases3-stripFullBoth-arr-inlined-bad	63	10	44	2.51	4	41	3	0
52	VERISEC.OpenSER_cases3-stripFullEnd-arr-bad	56	10	31	0.28	1	30	1	0
53	VERISEC.OpenSER_cases3-stripFullEnd-arr-inlined-bad	57	10	36	0.41	1	33	3	0
54	VERISEC.OpenSER_cases3-stripFullStart-arr-bad	58	10	31	1.84	3	30	1	0
55	VERISEC.OpenSER_cases3-stripFullStart-arr-inlined-bad	59	10	36	1.74	2	33	3	0
56	VERISEC.OpenSER_cases3-stripNone-arr-bad	52	10	23	0.14	1	22	1	0
57	VERISEC.OpenSER_cases3-stripNone-arr-inlined-bad	53	10	28	0.21	1	25	3	0
58	VERISEC.OpenSER_cases3-stripSpacesBoth-arr-bad	58	10	29	1.3	3	28	1	0
59	VERISEC.OpenSER_cases3-stripSpacesBoth-arr-inlined-bad	59	10	34	1.5	3	31	3	0
60	VERISEC.OpenSER_cases3-stripSpacesEnd-arr-bad	55	10	26	0.24	1	25	1	0

Table B.10: Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part II.

	Module	L	B	P	Time		Properties		
					Solver	Total	Passed	Violated	Fail
61	VERISEC.OpenSER_cases3-stripSpacesEnd-arr-inlined-bad	56	10	31	0.3	1	28	3	0
62	VERISEC.OpenSER_cases3-stripSpacesStart-arr-bad	55	10	26	1.11	1	25	1	0
63	VERISEC.OpenSER_cases3-stripSpacesStart-arr-inlined-bad	56	10	31	1.29	3	28	3	0
64	VERISEC.samba_simp-bad	22	1	4	0	1	4	1	0
65	VERISEC.sendmail_both-bad	44	5	15	0.03	1	13	2	0
66	VERISEC.sendmail_close-angle-ptr-no-test-bad	45	3	8	0.01	1	7	1	0
67	VERISEC.sendmail_inner-bad	37	4	9	0	1	8	1	0
68	VERISEC.sendmail_mime7to8-arr-one-char-heavy-test-bad	48	10	15	0.22	1	12	3	0
69	VERISEC.sendmail_mime7to8-arr-one-char-med-test-bad	46	10	15	0.18	1	10	5	0
70	VERISEC.sendmail_mime7to8-arr-one-char-no-test-bad	29	10	7	0.02	1	4	3	0
71	VERISEC.sendmail_mime7to8-arr-three-chars-med-test-bad	94	10	41	2.97	3	32	9	0
72	VERISEC.sendmail_mime7to8-ptr-one-char-heavy-test-bad	46	10	14	0.45	1	11	3	0
73	VERISEC.sendmail_mime7to8-ptr-three-chars-med-test-bad	86	10	36	4.14	5	25	11	0
74	VERISEC.sendmail_mime7to8-ptr-three-chars-no-test-bad	42	10	15	0.05	1	8	7	0
75	VERISEC.sendmail_outer-bad	43	4	15	0.02	1	15	1	0
76	VERISEC.sendmail_prescan-arr-med-test-bad	86	5	21	0.06	1	20	1	0
77	VERISEC.sendmail_prescan-arr-min-test-bad	83	5	21	0.05	1	20	1	0
78	VERISEC.sendmail_tTflag-arr-one-loop-bad	23	11	9	0.22	1	6	3	0
79	VERISEC.sendmail_util-bad	136	30	33	39.67	52	26	7	0
80	VERISEC.SpamAssassin_loop-bad	45	7	33	2.58	3	32	1	0
81	VERISEC.wu-ftpd_simple-bad	53	4	14	0	1	13	1	0
82	VERISEC.wu-ftpd_small-invalid	44	4	14	0	1	10	4	0
83	VERISEC.wu-ftpd_strcpy-strcat-bad	63	5	29	0.02	1	27	2	0
-	**Total**	**4569**	-	**2024**	**127**	**226**	**1808**	**216**	**0**

TABLE B.11: Results of applying ESBMC to the verification of the bad benchmarks from the VERISEC suite - Part III.

B.6 WCET Suite

Table B.12 shows the results of applying ESBMC to the verification of the programs from the WCET suite. Note that ESBMC finds four property violations in two programs (*duff_nondet* and *st*), which we inspected manually.

					Time		Properties		
	Module	*L*	B	*P*	Solver	Total	Passed	Violated	Fail
1	WCET.cnt	133	11	27	0.144	1	27	0	0
2	WCET.cover	238	121	196	0.067	1	196	0	0
3	WCET.duff_det	86	101	39	0.026	1	39	0	0
4	***WCET.duff_nondet***	***86***	***101***	***38***	***0.052***	***1***	***37***	***1***	***0***
5	WCET.expint	157	101	33	0.016	1	33	0	0
6	WCET.fdct	238	9	314	0.091	1	314	0	0
7	WCET.ns_det	531	6	22	0.012	2	22	0	0
8	WCET.ns_nondet	531	6	22	5.296	14	22	0	0
9	WCET.statemate	1273	3	6	0.25	1	6	0	0
10	***WCET.st***	***157***	***1001***	***29***	***1.332***	***50***	***26***	***3***	***0***
-	**Total**	**3430**	-	**726**	**7**	**73**	**722**	**4**	**0**

Table B.12: Results of applying ESBMC to the verification of the benchmarks from the WCET suite.

Appendix C

Functions of the Pthread Library

ESBMC is able to model check multi-threaded programs that use some functions of the POSIX Pthread Library [135]. This appendix thus describes the main functions of the Pthread library that we support.

- **pthread_create()**: This function creates a new thread.
- **pthread_exit():** This function terminates the calling thread.
- **pthread_mutex_init():** This function initializes the mutex that is used for performing synchronization among the threads.
- **pthread_mutex_lock():** This function locks the mutex if it is unlocked; otherwise it blocks the current thread until the mutex is released and can then be locked successfully again.
- **pthread_mutex_unlock():** This function unlocks the mutex that was locked previously by the same thread.
- **pthread_rwlock_init():** This function initializes the read-write lock object, which allows concurrent read access to an object but requires exclusive access for write operations.
- **pthread_rwlock_trywrlock()**, **pthread_rwlock_wrlock():** These functions locks a read-write lock object for writing.
- **pthread_rwlock_unlock():** This function unlocks a read-write lock object.
- **pthread_cond_init():** This function initializes the condition variable.
- **pthread_cond_wait():** This function is used to block the thread on a condition variable and the blocked thread is awakened only if another thread calls signal or broadcast.

- **pthread_cond_signal():** If there are several threads that are blocked on a condition variable, then this function unblocks at least one of them (but there is no guarantee of which one will be woken up due to the scheduling policy).
- **pthread_cond_broadcast():** This function unblocks all threads currently blocked on the specified condition variable.
- **pthread_cond_destroy():** This function destroys the given condition variable, i.e., the object becomes uninitialized.

Appendix D

Counterexample

This appendix shows the counterexample that is generated for the multi-threaded program shown in Figure 4.17.

```
Counterexample:

State 1 file /usr/include/time.h line 275 thread 0
----------------------------------------------------
  __tzname={ NULL, NULL }

State 2 file /usr/include/time.h line 276 thread 0
----------------------------------------------------
  __daylight=0 (00000000000000000000000000000000)

State 3 file /usr/include/time.h line 277 thread 0
----------------------------------------------------
  __timezone=0 (00000000000000000000000000000000)

State 4 file /usr/include/time.h line 282 thread 0
----------------------------------------------------
  tzname={ NULL, NULL }

State 5 file <builtin-library> line 41 function pthread_mutex_lock thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=TRUE

State 6 file <builtin-library> line 42 function pthread_mutex_lock thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::deadlock_mutex=FALSE

State 7 file <builtin-library> line 43 function pthread_mutex_lock thread 0
----------------------------------------------------
  trds_in_run=0 (00000000000000000000000000000000)

State 8 file <builtin-library> line 43 function pthread_mutex_lock thread 0
----------------------------------------------------
  trds_count=0 (00000000000000000000000000000000)

State 9 file <builtin-library> line 43 function pthread_mutex_lock thread 0
```

```
----------------------------------------------------
  count_lock=0 (00000000000000000000000000000000)

State 10 file carter01_bad.c line 2 thread 0
----------------------------------------------------
  m={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 11 file carter01_bad.c line 2 thread 0
----------------------------------------------------
  l={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 12 file carter01_bad.c line 3 thread 0
----------------------------------------------------
  A=0 (00000000000000000000000000000000)

State 13 file carter01_bad.c line 3 thread 0
----------------------------------------------------
  B=0 (00000000000000000000000000000000)

State 14 file <builtin-library> line 316 function pthread_cond_wait thread 0
----------------------------------------------------
  count_wait=0 (00000000000000000000000000000000)

State 15 file <builtin-library> line 317 function pthread_cond_wait thread 0
----------------------------------------------------
  <built-in-library>::pthread_cond_wait::1::deadlock_wait=FALSE

State 16 file <built-in> line 12 thread 0
----------------------------------------------------
  __ESBMC_alloc=(assignment removed)

State 17 file <built-in> line 13 thread 0
----------------------------------------------------
  __ESBMC_alloc_size=(assignment removed)

State 18 file <built-in> line 19 thread 0
----------------------------------------------------
  __ESBMC_rounding_mode=0 (00000000000000000000000000000000)

State 21 file carter01_bad.c line 31 function main thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_init::mutex=&m

State 22 file carter01_bad.c line 31 function main thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_init::mutexattr=NULL

State 23 file carter01_bad.c line 31 function main thread 0
----------------------------------------------------
  m={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 24 file carter01_bad.c line 31 function main thread 0
----------------------------------------------------
```

```
  m={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 25 file carter01_bad.c line 31 function main thread 0
----------------------------------------------------
  m={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 28 file carter01_bad.c line 32 function main thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_init::mutex=&l

State 29 file carter01_bad.c line 32 function main thread 0
----------------------------------------------------
  <built-in-library>::pthread_mutex_init::mutexattr=NULL

State 30 file carter01_bad.c line 32 function main thread 0
----------------------------------------------------
  l={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 31 file carter01_bad.c line 32 function main thread 0
----------------------------------------------------
  l={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 32 file carter01_bad.c line 32 function main thread 0
----------------------------------------------------
  l={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 51 file carter01_bad.c line 5 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::mutex=&m

State 52 file carter01_bad.c line 5 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=TRUE

State 54 file carter01_bad.c line 5 function t1 thread 1
----------------------------------------------------
  m={ .__data={ .__lock=1, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 59 file carter01_bad.c line 6 function t1 thread 1
----------------------------------------------------
  A=1 (00000000000000000000000000000001)

State 62 file carter01_bad.c line 7 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::mutex=&l

State 63 file carter01_bad.c line 7 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=TRUE
```

```
State 65 file carter01_bad.c line 7 function t1 thread 1
----------------------------------------------------
  l={ .__data={ .__lock=1, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 71 file carter01_bad.c line 8 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_unlock::mutex=&m

State 72 file carter01_bad.c line 8 function t1 thread 1
----------------------------------------------------
  m={ .__data={ .__lock=0, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 75 file carter01_bad.c line 10 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::mutex=&m

State 79 file carter01_bad.c line 16 function t2 thread 2
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::mutex=&m

State 80 file carter01_bad.c line 16 function t2 thread 2
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=TRUE

State 82 file carter01_bad.c line 16 function t2 thread 2
----------------------------------------------------
  m={ .__data={ .__lock=1, .__count=0, .__owner=0, .__kind=0, .__nusers=0,
    .__list={ .__next=NULL }, .__spins=0 } }

State 87 file carter01_bad.c line 17 function t2 thread 2
----------------------------------------------------
  B=1 (00000000000000000000000000000001)

State 90 file carter01_bad.c line 18 function t2 thread 2
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::mutex=&l

State 91 file carter01_bad.c line 10 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=FALSE

State 93 file carter01_bad.c line 10 function t1 thread 1
----------------------------------------------------
  count_lock=1 (00000000000000000000000000000001)

State 96 file carter01_bad.c line 10 function t1 thread 1
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::deadlock_mutex=FALSE

State 105 file carter01_bad.c line 18 function t2 thread 2
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::unlocked=FALSE

State 107 file carter01_bad.c line 18 function t2 thread 2
```

```
----------------------------------------------------
  count_lock=2 (00000000000000000000000000000010)

State 110 file carter01_bad.c line 18 function t2 thread 2
----------------------------------------------------
  <built-in-library>::pthread_mutex_lock::1::deadlock_mutex=TRUE

Violated property:
  file carter01_bad.c line 18 function t2
  deadlock detected with mutex lock
  !deadlock_mutex
```

References

[1] The economic impact of inadequate infrastructure for software testing. *Technical Planning Report 02-3, National Institute of Standards and Technology*, 2002.

[2] *MiBench Version 1.0*. http://www.eecs.umich.edu/mibench/, 2009.

[3] Common vulnerabilities and exposures. In *http://cve.mitre.org/*, 2010.

[4] *Advanced Linux Sound Architecture*. http://www.alsa-project.org/, 2011.

[5] *DirectFB*. http://directfb.org/, 2011.

[6] *Television with Linux*. http://www.linuxtv.org/, 2011.

[7] Accellera. *Property Specification Language (Reference Manual)*. Available at http://www.eda.org/vfv/docs/PSL-v1.1.pdf, 2004.

[8] Torben Amtoft and Anindya Banerjee. Verification condition generation for conditional information flow. In *FMSE*, pages 2–11, 2007.

[9] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[10] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, *LNCS* 3925, pages 146–162, 2006.

[11] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, 2009.

[12] Domagoj Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[13] Domagoj Babić and Alan J. Hu. Calysto: Scalable and Precise Extended Static Checking. In *ICSE*, pages 211–220, 2008.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[15] Subhashini Balakrishnan and Sofiene Tahar. On the formal verification of embedded software using multiway decision graphs. *Technical Report TR-402, Concordia University, Montreal, Canada*, 1997.

[16] Thomas Ball and Sriram K. Rajamani. *SLIC: A Specification Language for Interface Checking (of C)*. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[17] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs 0002, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. In *VSTTE*, *LNCS* 4171, pages 144–152, 2005.

[18] Michael Barnett and K. Rustan M. Leino. To goto where no statement has gone before. In *VSTTE*, *LNCS* 6217, pages 157–168, 2010.

[19] Clark Barrett, Leonardo de Moura, and Aaron Stump SMT-COMP: Satisfiability Modulo Theories Competition. In *CAV*, *LNCS* 3576, pages 20–23, 2005.

[20] Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, *LNCS* 4590, pages 298–302, 2007.

[21] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *STTT*, 9(5-6):505–525, 2007.

[22] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *CoRR*, abs/cs/0611029, 2006.

[23] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[24] Armin Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[25] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, *LNCS* 1579, pages 193–207, 1999.

[26] Nikolaj Bjørner and Leonardo de Moura. $Z3^{10}$: Applications, enablers, challenges and directions. In *Sixth International Workshop on Constraints in Formal Verification*, 2009.

[27] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL constructs for MathSAT: a preliminary report. *Electr. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

[28] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *TACAS*, *LNCS* 3440, pages 317–333, 2005.

[29] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[30] Daniel Brand. Verification of large synthesized designs. In *ICCAD*, pages 534–537, 1993.

[31] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, *LNCS* 5505, pages 174–177, 2009.

[32] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *CAV*, *LNCS* 5123, pages 299–303, 2008.

[33] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, 1990.

[34] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.

[35] Sagar Chaki, Edmund M. Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates. In *CHARME*, *LNCS* 2860, pages 19–34, 2003.

[36] Marsha Chechik. *Personal communication.* 2011.

[37] Alonzo Church. A note on the entscheidungsproblem. *Journal of Symbolic Logic*, 1:40-41, 1936.

[38] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *CAV*, *LNCS* 1633, pages 495–499, 1999.

[39] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: a software model checking approach. In *FMCAD*, 2010.

[40] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking.* MIT Publishers, 2000.

[41] Edmund M. Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *ASP-DAC*, pages 308–311, 2003.

[42] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, *LNCS* 2988, pages 168–176, 2004.

[43] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design*, 25:105–127, 2004.

[44] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS 2005*, *LNCS* 3440, pages 570–574, 2005.

[45] Edmund M. Clarke, Daniel Kroening, Ofer Strichman, and Joel Ouaknine. Completeness and complexity of bounded model checking. In *VMCAI*, *LNCS* 2937, pages 85–96, 2004.

[46] Edmund M. Clarke. SAT-based counterexample guided abstraction refinement in model checking. In *CADE*, *LNCS* 2741, page 1, 2003.

[47] Edmund M. Clarke, Anubhav Gupta, Himanshu Jain, and Helmut Veith. Model checking: Back and forth between hardware and software. In *VSTTE*, pages 251–255, 2005.

[48] James A. Clause and Alessandro Orso. Leakpoint: pinpointing the causes of memory leaks. In *ICSE (1)*, pages 515–524, 2010.

[49] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *CAV*, *LNCS* 3576, pages 296–300, 2005.

[50] Lucas Cordeiro and Bernd Fischer. Bounded model checking of multi-threaded software using smt solvers. In *Presentation-only paper at 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*, 2010.

[51] Lucas Cordeiro, Raimundo Barreto, Rafael Barcelos, Meuse Oliveira, Vicente Lucena Jr., and Paulo Maciel. Txm: An agile hw/sw development methodology for building medical devices. *In ACM SIGSOFT Software Engineering Notes.*, 32(6):32, 2007.

[52] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. *Efficient SMT-based Bounded Model Checker (ESBMC).* users.ecs.soton.ac.uk/lcc08r/esbmc, 2009.

[53] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *ASE*, pages 137–148, 2009.

[54] Lucas Cordeiro, Bernd Fischer, and João Marques-Silva. Continuous verification of large embedded software using SMT-based bounded model checking. In *ECBS*, pages 160–169, 2010.

[55] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[56] Leonardo de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In *LPAR Workshops*, 2008.

[57] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, *LNCS* 4963, pages 337–340, 2008.

[58] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: An appetizer. In *SBMF*, *LNCS* 5902, pages 23–36, 2009.

[59] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE*, *LNCS* 2392, pages 438–455, 2002.

[60] Eva Dejnozkova and Petr Dokladal. Asynchronous multi-core architecture for level set methods. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1–4, 2004.

[61] Jayant DeSouza, Bob Kuhn, Bronis R. de Supinski, Victor Samofalov, Sergey Zheltov, and Stanislav Bratanov. Automated, scalable debugging of MPI programs with intel®message checker. In *SE-HPCS*, pages 78–82, 2005.

[62] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[63] Alastair Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, *LNCS* 6015, pages 280–295, 2010.

[64] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[65] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper, *http://yices.csl.sri.com/documentation.shtml*, 2009.

[66] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[67] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232, 2000.

[68] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[69] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.

[70] Martin Fowler. Continuous Integration. ThoughtWorks. *http://martinfowler.com*, 2006.

[71] Malay K. Ganai and Aarti Gupta. Accelerating high-level bounded model checking. In *ICCAD*, pages 794–801, 2006.

[72] Malay K. Ganai and Aarti Gupta. Completeness in SMT-based BMC for software programs. In *DATE*, pages 831–836, 2008.

[73] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN*, *LNCS* 5156, pages 114–133, 2008.

[74] Naghmeh Ghafari, Alan Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: An empirical evaluation. In *SPIN*, *LNCS* 6349, pages 227–244, 2010.

[75] Patrice Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-explosion Problem.* University of Liege, PhD thesis, 1995.

[76] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.

[77] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[78] Benny Godlin and Ofer Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[79] H. Goldstein. Checking the play in plug-and-play. *Spectrum, IEEE*, 39(6):50–55, 2002.

[80] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Program. Lang. Syst.*, 2(4):564–579, 1980.

[81] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Fault localization using a model checker. *Softw. Test., Verif. Reliab.*, 20(2):149–173, 2010.

[82] Alex Groce, Klaus Havelund, and Margaret H. Smith. From scripts to specifications: the evolution of a flight software testing effort. In *ICSE (2)*, pages 129–138, 2010.

[83] Formal Methods Group. *SymC.* http://www-ti.informatik.uni-tuebingen.de/ fmg/symc/, 2008.

[84] Orna Grumberg, Flavio Lerda, Ofer Strichman, and Michael Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL*, pages 122–131, 2005.

[85] Elsa L. Gunter and Doron Peled. Model checking, testing and verification working together. *Formal Asp. Comput.*, 17(2):201–221, 2005.

[86] Sumit Gupta. *High Level Synthesis Benchmarks Suite.* http://mesl.ucsd.edu/spark/benchmarks.shtml, 2009.

[87] Eclipse Helios. Eclipse IDE for C/C++ developers, 2010.

[88] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Dirk Beyer. *BLAST: Berkeley Lazy Abstraction Software Verification Tool.* http://mtc.epfl.ch/software-tools/blast/, 2009.

[89] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *CAV*, *LNCS* 2725, pages 262–274, 2003.

[90] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[91] Gerard J. Holzmann. The Spin Model Checker - Primer and Reference Manual. *Addison-Wesley*, 2003.

[92] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the Spin model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[93] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the Swarm tool. In *SPIN*, *LNCS* 5156, pages 134–143, 2008.

[94] Michael Huth and Mark Ryan Logic in Computer Science: modelling and reasoning about systems. *Cambridge University Press*, 2004.

[95] ISO. ISO/IEC 9899:1999: Programming languages C. *International Organization for Standardization*, 1999.

[96] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai Model checking C programs using F-SOFT. *ICCD*, pages 297–308, 2005.

[97] Franjo Ivancic. *Personal communication.* 2011.

[98] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT solvers to verify high-integrity programs. In *AFM*, pages 60–68, 2007.

[99] Paul B. Jackson and Grant Olney Passmore. *Proving SPARK Verification Conditions with SMT solvers.* Technical Report, University of Edinburgh, 2009.

[100] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[101] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theor. Comput. Sci.*, 167(1&2):47–72, 1996.

[102] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic reduction of thread interleavings in concurrent programs. In *TACAS*, *LNCS* 5505, pages 124–138, 2009.

[103] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, *LNCS* 5643, pages 398–413, 2009.

[104] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2002.

[105] Daniel Kroening. *Personal communication*. 2009.

[106] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC 2003*, pages 368–371, 2003.

[107] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Technical Report, CMU-CS-03-126*, 2003.

[108] Daniel Kroening and Sanjit A. Seshia. Formal verification at higher levels of abstraction. In *ICCAD*, pages 572–578, 2007.

[109] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.

[110] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392, 2007.

[111] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV*, *LNCS* 5643, pages 509–524, 2009.

[112] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[113] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[114] D. Lettnin, P. K. Nalla, J. Ruf, R. Weiss, A. Braun, J. Gerlach, T. Kropf, and W. Rosenstiel. Semiformal verification of temporal properties in embedded software. *GI/ITG/GMM Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Erlangen, Germany*, 2007.

[115] Djones Lettnin, Pradeep Kumar Nalla, Jörg Behrend, Jürgen Ruf, Joachim Gerlach, Thomas Kropf, Wolfgang Rosenstiel, Volker Schönknecht, and Stephan Reitemeyer. Semiformal verification of temporal properties in automotive hardware dependent software. In *DATE*, pages 1214–1217, 2009.

[116] Sung-Soo Lim. *SNU Real-Time Benchmarks Suite.* http://archi.snu.ac.kr/realtime/benchmark/, 2009.

[117] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, 36(1):329–339, 2008.

[118] James R. Lyle and David W. Binkley. Program slicing in the presence of pointers. In *Third Annual Software Engineering Research Forum*, pages 11–12, 1993.

[119] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426, 2007.

[120] Nicolas Markey and Ph. Schnoebelen. Symbolic model checking for simply-timed systems. In *FORMATS/FTRTFT*, pages 102–117, 2004.

[121] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *ISQED*, pages 370–375, 2006.

[122] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *ASE*, pages 128–137, 2008.

[123] John McCarthy. Towards a mathematical science of computation. In *In IFIP*, pages 21–28, 1962.

[124] Kenneth L. McMillan. *The Cadence SMV Model Checker.* http://www.kenmcmil.com/smv.html, 2010.

[125] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, *LNCS* 2725, pages 1–13, 2003.

[126] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *TACAS*, *LNCS* 3440, pages 1–12, 2005.

[127] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, *LNCS* 4144, pages 123–136, 2006.

[128] Kenneth L. McMillan. Interpolants and symbolic model checking. In *VMCAI*, *LNCS* 4349, pages 89–90, 2007.

[129] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, *LNCS* 2619, pages 2–17, 2003.

[130] Elliott Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 2009.

[131] José Vander Meulen and Charles Pecheur. Combining partial order reduction with bounded model checking. In *Communicating Process Architectures (CPA)*, pages 29–48, 2009.

[132] Jeremy Morse. *Kerberos Git*. https://www.studentrobotics.org/trac/wiki/Kerberos/Git, 2011.

[133] Mohammad Reza Mousavi and Michel Reniers. A congruence rule format with universal quantification. *Electron. Notes Theor. Comput. Sci.*, 192(1):109–124, 2007.

[134] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.

[135] Frank Mueller. A library implementation of posix threads under unix. In *USENIX*, pages 29–41, 1993.

[136] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[137] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *PLDI*, pages 362–371, 2008.

[138] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[139] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[140] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *CC*, *LNCS* 2304, pages 213–228, 2002.

[141] NXP. *High definition IP and hybrid DTV set-top box STB225*. http://www.nxp.com/, 2009.

[142] Tom Ostrand. *Siemens Corporate Research*. http://sir.unl.edu/portal/, 2010.

[143] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers, 2011.

[144] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *ICSE (1)*, pages 245–254, 2010.

[145] Jacques Patarin and Louis Goubin. Trapdoor one-way permutations and multivariate polynominals. In *ICICS*, *LNCS* 1334, pages 356–368. Springer, 1997.

[146] Doron Peled. All from one, one for all: on model checking using representatives. In *CAV*, *LNCS* 697, pages 409–423, 1993.

[147] Doron Peled. Model checking and testing combined. In *ICALP*, *LNCS* 2719, pages 47–63, 2003.

[148] Lorenzo Platania. *Eureka Benchmark Suite*. http://www.ai-lab.it/eureka/bmc.html, 2009.

[149] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[150] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS*, *LNCS* 3440, pages 93–107, 2005.

[151] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[152] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *CAV*, *LNCS* 3576, pages 82–97, 2005.

[153] Muralikrishna Ramanathan. *flex*. http://sir.unl.edu/portal/, 2010.

[154] John A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.

[155] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[156] Neha Rungta and Eric G. Mercer. Clash of the titans: tools and techniques for hunting bugs in concurrent programs. In *PADTAD*, pages 1–10, 2009.

[157] Sriram Sankaranarayanan. *NECLA Static Analysis Benchmarks*. http://www.nec-labs.com/research/system/, 2009.

[158] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[159] Jeff Scott, Lea Hwang Lee, Ann Chin, John Arends, and Bill Moyer. Designing the low-power m*core architecture. In *ICCD*, pages 94–101, 1999.

[160] Koushik Sen. Concolic testing. In *ASE*, pages 571–572. ACM, 2007.

[161] Koushik Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.

[162] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, *LNCS* 1954, pages 108–125, 2000.

[163] Joao P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[164] SMT-LIB. *The Satisfiability Modulo Theories Library.* http://combination.cs.uiowa.edu/smtlib, 2009.

[165] Fabio Somenzi and Roderick Bloem. Efficient buechi automata from LTL formulae. In *CAV*, *LNCS* 1855, page 247263, 2000.

[166] Ian Sommerville. *Software Engineering.* Pearson Education Limited, 2007.

[167] Ofer Strichman. Regression verification: Proving the equivalence of similar programs. In *CAV*, *LNCS* 5643, page 63, 2009.

[168] Aaron Stump and Morgan Deters. *Satisfiability Modulo Theories Competition.* http://www.smtcomp.org/, 2010.

[169] Andrew S. Tanenbaum. *Computer networks: 4th edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[170] Olivier Thiry and Luc J. Claesen A formal verification technique for embedded software. *ICCD*, pages 352–357, 1996.

[171] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, *LNCS* 5643, pages 477–492, 2009.

[172] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *In J. Siekmann and G. Wrightson, editors, Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970.*

[173] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, *LNCS* 1043, pages 238–266, 1996.

[174] Alberto L. Sangiovanni-Vincentelli, Luca P. Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. *DAC*, pages 409–414, 2004.

[175] Nguyen Le Vinh. *The Flasher Manager Application.* http://users.polytech.unice.fr/ rueher/Benchs/FM/, 2010.

[176] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[177] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, *LNCS* 6015, pages 328–342, 2010.

[178] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, *LNCS* 4963, pages 382–396, 2008.

[179] Christoph M. Wintersteiger. *Compiling GOTO-Programs.* http://www.cprover.org/goto-cc/, 2009.

[180] Yichen Xie and Alex Aiken. Scalable error detection using Boolean satisfiability. *SIGPLAN Not.*, pages 351–363, 2005.

[181] Liang Xu. SMT-based bounded model checking for real-time systems. In *QSIC*, pages 120–125, 2008.

[182] Yu Yang. *Inspect: A Framework for Dynamic Verification of Multithreaded C Programs.* http://www.cs.utah.edu/ yuyang/inspect/, 2010.

[183] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert Kirby. Runtime model checking of multithreaded C/C++ programs. In *Technical Report, UUCS-07-008*, 2007.

[184] Aleks Zaks, Ilya Shlyakhter, Franjo Ivancic, Srihari Cadambi, Zijiang Yang, Malay Ganai, Aarti Gupta, and Pranav Ashar. Using range analysis for software verification. In *4th International Workshop on Software Verification and Validation*, 2006.

[185] C. Păsăreanu, P. Mehlitz, D. Bushnell, G. Burlet. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software In *ISSTA*, pages 15–26, 2008.