Proceedings of the 6th International Workshop on Systems Software Verification (SSV 2011)

Jörg Brauer Marco Roveri Hendrik Tews
(Editors)

Nijmegen, 26 August 2011

Technical report TUD–FI11–02–August 2011, TU Dresden

Preface

Industrial-strength software analysis and verification has advanced in recent years through the introduction of model checking, automated and interactive theorem proving, and static analysis techniques, as well as correctness by design, correctness by contract, and model-driven development. However, many techniques are working under restrictive assumptions that are invalidated by complex embedded systems software such as operating system kernels, low-level device drivers, or micro-controller code.

The aim of SSV workshop series is to bring together researchers and developers from both academia and industry who are facing real software and real problems with the goal of finding real, applicable solutions. It has always been the goal of SSV program committees to let "real problem" really mean real problem (in contrast to real academic problem).

The 6th SSV workshop was held on August 26 in Nijmegen in the Netherlands. The workshop was co-located with the second conference on Interactive Theorem Proving (ITP 2011), which took place from 22–25 August at the same place.

The program chairs and organization committee of SSV 2011 have been

Jörg Brauer, RWTH Aachen University, Germany
Marco Roveri, FBK-irst, Italy
Hendrik Tews, TU Dresden, Germany

The SSV program chairs gratefully acknowledge the sponsorship of National ICT Australia Ltd (NICTA), Australia's Information and Communications Technology Research Centre of Excellence, and of the Ultra high speed mobile information and communication (UMIC) cluster of excellence at RWTH Aachen University in Germany.

August 9, 2011 Jörg Brauer, Marco Roveri and Hendrik Tews

Table of Contents

Refinement-based CFG Reconstruction from Executables [⋆,⋆⋆]

Sébastien Bardin, Philippe Herrmann, and Franck Védrine

CEA, LIST,
Gif-sur-Yvette CEDEX, 91191 France
`first.name@cea.fr`

Abstract. We address the issue of recovering a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. The problem is tackled in an original way, with a refinement-based static analysis working over finite sets of constant values. Requirement propagation allows the analysis to automatically adjust the domain precision only where it is needed, resulting in precise CFG recovery at moderate cost. First experiments, including an industrial case study, show that the method outperforms standard analyses in terms of precision, efficiency or robustness.

Motivation. Automatic analysis of programs from their executable files has many potential applications in safety and security, for example: automatic analysis of mobile code and malware, security testing or worst case execution time estimation. We address the problem of (safe) CFG reconstruction, i.e. constructing a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. CFG reconstruction is a cornerstone of safe binary-level analysis: if the recovery is unsafe, subsequent analyses will be unsafe too; if it is too rough, they will be blurred by too many unfeasible branches and instructions.

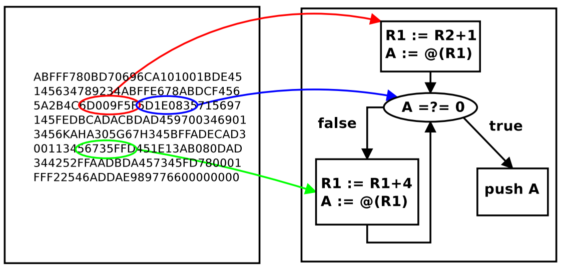
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F6D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
R1 := R2+1
A := @(R1)
A =?= 0
false
true
R1 := R1+4
A := @(R1)
push A

Fig. 1. CFG reconstruction from an executable file

Challenges. Such an approximation is difficult to obtain mainly because of dynamic jumps, i.e. jump instructions whose target expression is resolved at run-time and may

⋆ Work partially funded by ANR (grants ANR-05-RNTL-02606 and ANR-08-SEGI-006).

⋆⋆ The material presented here is taken from a preliminary version of the VMCAI'11 paper [3].

vary from one execution to the other. Dynamic jumps are very sensitive instructions and a small loss in precision on target expressions may affect dramatically the quality of the subsequent analysis, leading to vicious circles between value analysis and CFG reconstruction. Moreover, there is no reason why all valid targets of a dynamic jump should follow a nice regular pattern. Indeed they are just addresses in the executable code, often arbitrarily assigned by a compiler. Hence any analysis based on popular domains (i.e. convex domains possibly enhanced with congruence information) will introduce many false targets. For example, consider an instruction `cgoto(x)` with `x` $\in \{1355, 1356, 2126\}$: such an analysis cannot recover better than `x` $\in [1355..2126]$, reporting 99% of false targets.

Note that, unfortunately, dynamic jumps are ubiquitous in native code programs: they are introduced at compile-time either for efficiency (`switch` in C) or by necessity (return statements, function pointers in C, virtual methods in C++, etc.).

Related approaches. Industrial tools like IDA PRO [10] or AIT [9] usually rely on linear sweep decoding (brute force decoding of all code addresses) or recursive traversal (recursive decoding until a dynamic jump is encountered), enhanced with limited constant propagation, pattern matching techniques based on the knowledge of the compiling chain process and user annotations. These techniques are unsafe on general programs, missing many legal targets and branches. The only safe techniques are those by Reps *et al.* [4, 5] - based mainly on stride intervals propagation, and by Kinder and Veith [7, 8] - based on k-set (sets of bounded cardinality) propagation. Experiments reported by the authors show that while each approach performs much better than current industrial tools, both techniques still recover many false targets. Especially, stride intervals cannot capture precisely sets of jump targets, and k-sets are too sensitive to their cardinality bound, potentially leading to either imprecise or expensive analyses.

Our approach. We propose an original refinement-based procedure to solve CFG reconstruction [3]. The procedure is built on two main steps: a forward k-set propagation with local cardinality bounds (ranging from 0 up to a given parameter $Kmax$), and a refinement step controlling these cardinality bounds.

The forward propagation is mostly a standard one, enhanced with a few original mechanisms: (1) abstract values are downcast according to local cardinality bounds, permitting to lose information and increase efficiency; (2) $\top$ values (i.e. abstract values denoting the whole domain) are tagged with additional information recording their origin (for example $\top_{\langle 1,3,12 \rangle}$ denotes the abstraction to $\top$ of the k-set $\{1, 3, 12\}$), allowing to pinpoint the *initial sources of precision loss* (ispl) and give clue for correction (cf. refinement); (3) alias, jump targets and branches that have been fired during propagation are recorded into a *journal* (cf. refinement).

Refinement is lazy and on-demand. When a jump expression evaluates to $\top$, the refinement mechanism takes place, trying to find out ispls responsible for the violation (guided by backward data dependencies and journal information) and to correct them by locally improving the domain precisions (using $\top$-flags).

Results. From a theoretical point of view, the procedure is sound and runs in polynomial-time. Moreover it is as precise as standard k-set propagation on a class of non-trivial programs, including dynamic jumps and alias [3]. From a practical point of view, the procedure has been implemented and evaluated on an industrial safety-critical program (32 kloc) and on small handcrafted programs. It appears to be reasonably efficient (taking less than 5 minutes for the industrial case study), very precise (only 7% of false targets, beating standard approaches based on convex domains by several orders of magnitude), and very robust: the procedure does need an initial parameter, but its exact value does not seem to matter.

References

1. Balakrishnan, G., Gruian, R., Reps, T. W., Teitelbaum, T.: CodeSurfer/x86-A Platform for Analyzing x86 Executables. In: CC 2005. Springer, Heidelberg (2005)
2. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
4. Balakrishnan, G., Reps, T. W.: Analyzing memory accesses in x86 executables. In: CC 2004. Springer, Heidelberg (2004)
5. Balakrishnan, G., Reps, T. W.: Analyzing Stripped Device-Driver Executables. In: TACAS 2008. Springer, Heidelberg (2008)
6. Godefroid, P., Levin, M. Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008. The Internet Society (2008)
7. Kinder, J., Veith, H.: Jakstab: A Static Analysis Platform for Binaries. In: CAV 2008. Springer, Heidelberg (2008)
8. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: VMCAI 2009. Springer, Heidelberg (2009)
9. Absint homepage `http://www.absint.com/`
10. IDA Pro homepage `http://www.hex-rays.com/idapro`

VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java

Bart Jacobs

DistriNet Research Group
Department of Computer Science, Katholieke Universiteit Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract. VeriFast [6, 2, 3] is a verifier for single-threaded and multi-threaded C and Java programs annotated with preconditions and postconditions written in separation logic. To enable rich specifications, the programmer may define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract separation logic predicates. To enable verification of these rich specifications, the programmer may write lemma functions, i.e., functions that serve only as proofs that their precondition implies their postcondition. The verifier checks that lemma functions terminate and do not have side-effects. Verification proceeds by symbolic execution, where the heap is represented as a separation logic formula. Since neither VeriFast itself nor the underlying SMT solver do any significant search, verification time is predictable and low. The VeriFast IDE allows users to step through failed symbolic execution paths, and to inspect the symbolic state at each step. This yields a relatively comfortable interactive annotation authoring experience. We are currently using VeriFast to verify fine-grained concurrent data structures [1], unloadable kernel modules [5], and JavaCard programs [4].

Bibliography

[1] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[2] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, 2010.

[3] Bart Jacobs, Jan Smans, and Frank Piessens. The VeriFast program verifier: A tutorial. At http://www.cs.kuleuven.be/~bartj/verifast/, 2010.

[4] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java (invited paper). In *NASA Formal Methods Symposium (NFM)*, 2011.

[5] Bart Jacobs, Jan Smans, and Frank Piessens. Verification of unloadable modules. In *FM*, 2011.

[6] Bart Jacobs, Jan Smans, and Frank Piessens. VeriFast. Tool downloadable at http://www.cs.kuleuven.be/~bartj/verifast/, 2011.

Adaptable Value-Set Analysis for Low-Level Code

Jörg Brauer[1]*, René Rydhof Hansen[2], Stefan Kowalewski[1], Kim G. Larsen[2] and Mads Chr. Olesen[2]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Department of Computer Science, Aalborg University, Denmark

Abstract. This paper presents a framework for binary code analysis that uses only SAT-based algorithms. Within the framework, incremental SAT solving is used to perform a form of weakly relational value-set analysis in a novel way, connecting the expressiveness of the value sets to computational complexity. Another key feature of our framework is that it translates the semantics of binary code into an intermediate representation. This allows for a straightforward translation of the program semantics into Boolean logic and eases the implementation efforts, too. We show that leveraging the efficiency of contemporary SAT solvers allows us to prove interesting properties about medium-sized microcontroller programs.

1 Introduction

Model checking and abstract interpretation have long been considered as formal verification techniques that are diametrically opposed. In model checking, the behavior of a system is formally specified with a model. All paths through the system are then exhaustively checked against its requirements, which are classically specified in some temporal logic. Of course, the detailed nature of the requirements entails that the program is simulated in a fine-grained fashion, sometimes down to the level of individual bits. Since the complexity of this style of reasoning naturally leads to state explosion, there has thus been much interest in representing states symbolically so as to represent states that share some commonality without duplicating their commonality. As one instance, Boolean formulae have successfully been applied to this task [10].

By way of comparison, the key idea in abstract interpretation [14] is to abstract away from the detailed nature of states, and rather represent sets of concrete states using geometric concepts such as affine [19] or polyhedral spaces [15]. A program analyzer then operates over classes of states that are related in some sense — for example, sets of states that are contained by the shape of a convex polyhedron — rather than individual states. If the number of classes is small, then all paths through the program can be examined without incurring the problems of state explosion. Further, when carefully constructed, the classes of states can preserve sufficient information to prove correctness of

* The work of Jörg Brauer was carried out while being on leave at Aalborg University.

```
0x42 : ANDI R1 15
0x43 : ADD R0 R1
0x44 : LSL R0
0x45 : BRCS label
0x46 : INC R0
```

Fig. 1. The target of the conditional branch `BRCS label` depends on the carry flag after the left-shift; this value, in turn, depends on `R0` and `R1` on input.

the system. However, sometimes much detail is lost when working with abstract classes so that the technique cannot infer useful results; they are too imprecise. This is because the approach critically depends on the expressiveness of the classes and the class transformers chosen to model the instructions that arise in a program. It is thus desirable to express the class transformers, also called transfer functions, as accurately as possible. The difficulty of doing so, however, necessitates automation [24,30], especially if the programs/operations are low-level and defined over finite bit-vectors [5,6]. Recent research has demonstrated that automatic abstraction on top of sophisticated decision procedures provides a way to tame this complexity for low-level code [4,5,6,20,21,30]. Using these approaches, a decision procedure (such as a SAT or SMT solver) is invoked on a relational representation of the semantics of the program in order to automatically compute the desired abstraction. Since representing the concrete semantics as a Boolean formula has become a standard technique in program analysis (it is colloquially also referred to as *bit-blasting*), owing much to the advances of bounded model checking [11], such encodings can straightforwardly be derived.

1.1 Value-Set Analysis using SAT

This paper studies the algorithm of Barrett and King [4, Fig. 3], who showed how incremental SAT solving can be used to converge onto the non-relational value sets of a bit-vector characterized by a Boolean formula. When applying their technique to assembly code, however, the non-relational representation may be too imprecise. This is because blocks in assembly code frequently end in a conditional jump. This instruction, paired with the preceding ones, encodes certain constraints. For example, the 8-bit AVR code in Fig. 1 depends on two inputs `R0` and `R1`, which are used to mutate `R0` and `R1`. Control is transfered to `label` if the instruction `LSL R0` (logical left-shift of `R0`) sets the carry flag; otherwise, control proceeds with the increment located at address `0x46`.

To precisely approximate the value sets of `R0` at the entries and exits of each block, it is thus necessary to take the relation between the registers and the carry flag into account. For the values of `R0` in instruction `0x46`, e.g., one has to distinguish those inputs to the program which cause the carry flag to be set from those which lead to a cleared carry flag. To capture this relation, we argue that it is promising to consider a bit-vector representing not only `R0`, but simultaneously the carry flag (or any other status flag the branching instruction ultimately depends on). Suppose the initial block in Fig. 1 starting at address 0x42 is described by a Boolean formula φ. Our description relies on the convention that input bit-vectors are denoted $\boldsymbol{r0}$ and $\boldsymbol{r1}$, respectively, whereas the outputs are primed. Further, each bit-vector $\boldsymbol{r}$ takes the form $\boldsymbol{r} = \langle \boldsymbol{r}[0], \ldots, \boldsymbol{r}[7] \rangle$. Additionally, the carry flag on output is represented by a single propositional variable c'. Rather than projecting φ onto $\boldsymbol{r0}'$ for value-set analysis (VSA), one can likewise project φ onto the extended bit-vector $\langle \boldsymbol{r0}'[0], \ldots, \boldsymbol{r0}'[7], c' \rangle$. By decomposing the resulting value sets into those with c' cleared and those with c' set, we have reconstructed a 9-bit value-set representation for an 8-bit register that takes *some* relational information into account; it is thus *weakly relational.* The first contribution of this paper is a discussion and experimental evaluation of this technique, where status flags guide the extension of bit-vectors for VSA.

1.2 Intermediate Representation for Assembly Code

Implementing SAT-based program analyzers that operate on low-level representations requires significant effort because Boolean formulae for the entire instruction set of the hardware have to be provided. Although doing so is merely an engineering task, this situation is rather unsatisfactory if the program analyzer shall support different target platforms. Indeed, the instruction set of different hardware platforms often varies only in minor details, yet their sheer number makes the implementation (and testing, of course) complex. To overcome this complexity, we propose to decompose each instruction into an intermediate representation (IR) [3,9], where the instruction is characterized as an atomic sequence of basic operations. Each of the basic operations can then straightforwardly be translated into Boolean logic, thereby providing a representation that depends on few primitive operations only. We further elaborate on several characteristics of the IR and discuss our experiences with connecting VSA with Metamoc [17], a tool that performs worst-case execution time analysis using timed automata. In Metamoc, the system abstraction is generated on top of a static VSA.

1.3 Structure of the Presentation

To make this paper self-contained, Sect. 2 recapitulates the algorithm of Barrett and King [4]. The paper then builds towards the above mentioned contributions using a worked example in Sect. 3. The key ingredients of our framework are:

1. translate a given binary program into our IR,
2. express the semantics of the translated program in Boolean logic,
3. compute projections onto the relevant bit-vectors, and perform VSA using SAT solving until a fixed point is reached.

Each of these steps for the example program in Fig. 1 is discussed in its own subsection in Sect. 3. Then, Sect. 4 discusses an extension of the example to weak relations between different registers, before Sect. 5 presents some experimental evidence from our implementation. The paper concludes with a survey of related work in Sect. 6 and a discussion in Sect. 7.

2 Primer on Value-Set Analysis

The key idea of Barrett and King [4, Fig. 3] is to converge onto the value sets of a register using a form of dichotomic binary search [12]. Let φ denote a Boolean formula that characterizes a bit-vector $\boldsymbol{r}$. In the first iteration, the algorithm computes an over-approximation of the values of $\boldsymbol{r}$ by determining the least and greatest values $\boldsymbol{r}_\ell^1$ and $\boldsymbol{r}_u^1$ of $\boldsymbol{r}$ subject to φ. These values are obtained by repeatedly applying a SAT solver to φ in conjunction with blocking clauses. To illustrate the principle, consider determining $\boldsymbol{r}_\ell^1$. If the bit-vector $\boldsymbol{r}$ is w bits wide, the unsigned value of $\boldsymbol{r}$, denoted $\langle\!\langle \boldsymbol{r} \rangle\!\rangle = \sum_{i=0}^{w-1} 2^i \cdot \boldsymbol{r}[i]$, is in the range $0 \leq \langle\!\langle \boldsymbol{r} \rangle\!\rangle \leq 2^w - 1$, and so is $\langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle$. The constraint $0 \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \wedge \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^w - 1$ can be expressed disjunctively as $\mu_\ell \vee \mu_u$ where:

$$\mu_\ell = 0 \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^{w-1} - 1 \qquad\qquad \mu_u = 2^{w-1} \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^w - 1$$

To determine which of both disjuncts characterizes $\boldsymbol{r}_\ell^1$, it is sufficient to test the formula $\exists \boldsymbol{r} : \varphi \wedge \neg \boldsymbol{r}[w-1]$ for satisfiability. If satisfiable, then μ_ℓ is entailed by $\boldsymbol{r}_\ell^1$, and μ_u otherwise. This follows directly from the bit-vector representation of unsigned integer values. Suppose that $\exists \boldsymbol{r} : \varphi \wedge \boldsymbol{r}[w-1]$ is satisfiable, and thus $0 \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^{w-1} - 1$. We proceed by decomposing this refined characterization into a disjunction $\mu'_\ell \vee \mu'_u$ where

$$\mu'_\ell = 0 \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^{w-2} - 1 \qquad\qquad \mu'_u = 2^{w-2} \leq \langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq 2^{w-1} - 1$$

as above, and testing $\exists \boldsymbol{r} : \varphi \wedge \neg \boldsymbol{r}[w-1] \wedge \neg \boldsymbol{r}[w-2]$ for satisfiability. Repeating this step w times gives $\boldsymbol{r}_\ell^1$ exactly. We can likewise compute $\boldsymbol{r}_u^1$ and thus deduce:

$$\forall \boldsymbol{r} : \varphi \wedge (\langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r} \rangle\!\rangle \leq \langle\!\langle \boldsymbol{r}_u^1 \rangle\!\rangle)$$

The key idea of VSA is then to repeatedly apply this technique so as to alternatingly remove and add ranges from the initial interval $[\langle\!\langle \boldsymbol{r}_\ell^1 \rangle\!\rangle, \langle\!\langle \boldsymbol{r}_u^1 \rangle\!\rangle]$. It does so using alternating over- and under-approximation as follows. In the first iteration of the algorithm, the value set then contains all values in the computed range, i.e., $\mathcal{V}^1 = \{\boldsymbol{r}_\ell^1, \ldots, \boldsymbol{r}_u^1\}$. In the second iteration, the algorithm infers an over-approximate range of *non-solutions* and removes this range from $\mathcal{V}^1$. This gives an under-approximation of the actual value set of $\boldsymbol{r}$. To get this result, the algorithm computes the least and greatest non-solutions $\boldsymbol{r}_\ell^2$ and $\boldsymbol{r}_u^2$ within the range $\mathcal{V}^1$. The bounds are derived using dichotomic search based on $\neg\varphi$ rather than φ. An under-approximation of the value set of $\boldsymbol{r}$ is then obtained by eliminating $\{\boldsymbol{r}_\ell^2, \ldots, \boldsymbol{r}_u^2\}$ from $\mathcal{V}^1$, i.e., $\mathcal{V}^2 = \mathcal{V}^1 \setminus \{\boldsymbol{r}_\ell^2, \ldots, \boldsymbol{r}_u^2\}$. The under-approximation $\mathcal{V}^2$ is extended by adding an over-approximate range of solutions to $\mathcal{V}^2$. The algorithm thus proceeds by determining solutions $\boldsymbol{r}_\ell^3$ and $\boldsymbol{r}_u^3$ within the range $\boldsymbol{r}_\ell^2, \ldots, \boldsymbol{r}_u^2$. This turns the under-approximation $\mathcal{V}^2$ into an over-approximation $\mathcal{V}^3 = \mathcal{V}^2 \cup \{\boldsymbol{r}_\ell^3, \ldots, \boldsymbol{r}_u^3\}$, again followed by under-approximation. After a finite number k of iterations, no further solutions are found which are not contained in $\mathcal{V}^k$. The algorithm then terminates and $\mathcal{V}^k$ is the desired value set.

3 Worked Example

The key idea of our approach is to first translate each instruction in a program from a hardware-specific representation into an intermediate language. Liveness analysis is then performed to eliminate redundant (dead) operations from the IR. It turns out that liveness analysis is much more powerful in assembly code analysis than in traditional domains due to side-effects. Many instructions have side-effects, yet few of them actually influence the behavior of the program. The elimination of dead operation is followed by a conversion of each block into static single assignment (SSA) form [16]. The semantics of each block in the IR is then expressed in the computational domain of Boolean formulae. To derive over-approximations of the value sets of each register, we combine quantifier elimination using SAT solving [8] with VSA [29].

3.1 Translating a Binary Program

Recall that assembly instructions typically have side-effects. The instruction `ANDI R1 15` from Fig. 1, for instance, computes the bit-wise and of register `R1` with the constant `15` and stores the result in `R1` again. However, it also mutates some of the status flags, which are located in register `R95` (in case of the ATmega16). Our IR makes these *hidden* side-effects explicit, which then allows us to represent large parts of the instruction set using a small collection of building blocks. However, this additional flexibility also implies that some hardware-related information has to be included in the IR, most notably operand sizes and atomicity (instructions are executed atomically, and thus cannot be interrupted by an interrupt service routine). We tackle these two problems by representing a single instruction as an uninterruptible sequence of basic operations, and by postfixing the respective basic operation with one of the following operand-size identifiers:

Identifier	Meaning	Size	Example
`.b`	Bit	1	`XOR.b R0:0 R0:1 R0:2`
`.B`	Byte	8	`AND.B R0 R0 #15`
`.W`	Word	16	`INC.W R1 R1`
`.DW`	Double Word	32	`ADD.DW R0 R1 R2`

In this encoding, the first operand is always the target, followed by a varying number of source operands (e.g., bit-wise negation has a single source operand whereas addition has two). The AVR instruction `AND R0 #15` then translates into `AND.B R0 R0 #15`, thus far ignoring the side-effects. The side-effects are given in the instruction-set specification [1] by the following Boolean formula:

$$\begin{array}{llll} \boldsymbol{r95}'[1] \leftrightarrow \bigwedge_{i=0}^{7} \neg \boldsymbol{r0}'[i] & \wedge\, \boldsymbol{r95}'[2] \leftrightarrow & \boldsymbol{r0}'[7] & \wedge \\ \neg \boldsymbol{r95}'[3] & \wedge\, \boldsymbol{r95}'[4] \leftrightarrow & \boldsymbol{r95}'[2] \oplus \boldsymbol{r95}'[3] & \end{array}$$

Given the classical bit-wise operations, these side-effects are encoded (with some simplifications applied and using an additional macro `isZero`) as:

```
AND.B R1 R1 #15;    MOV.b R95:3 #0;          MOV.b R95:2 R0:7;
MOV.b R95:4 R95:2;  MOV.b R95:1 isZero(R0);
```

The other instructions can likewise be decomposed into such a sequence of building blocks, and then be conjoined to give a sequence that describes the instructions `0x42` to `0x45` from Fig. 1 as follows (note that some auxiliary variables are required to express the side-effects of `ADD R0 R1`):

```
0x42 : AND.B R1 R1 #15;   MOV.b R95:3 #0;          MOV.b R95:2 R1:7;
       MOV.b R95:4 R95:2; MOV.b R95:1 isZero(R1);
0x43 : MOV.B F R0;        ADD.B R0 R0 R1;          MOV.b R95:1 isZero(R0);
       MOV.b R95:2 R0:7;  XOR.b R95:4 R95:2 R95:3; AND.b R95:0 F:7 R1:7;
       NOT.b d R0:7;      AND.b e R1:7 d;          OR.b R95:0 R95:0 e;
       AND.b e d F:7;     OR.b R95:0 R95:0 e;      AND.b e F:7 R1:7;
       AND.b R95:3 e d;   NOT.b f F:7;             NOT.b g R1:7;
       AND.b f f g;       AND.b f f d;             OR.b R95:3 R95:3 f;
0x44 : MOV.b R95:5 R0:3;  MOV.b R95:0 R0:7;        LSL.B R0 R0 #1;
       MOV.b R95:2 R0:7;  XOR.b R95:3 R95:0 R95:2; XOR.b R95:4 R95:2 R95:3;
       MOV.b R95:1 isZero(R0);
0x45 : BRANCH (R95:0) label #0x46;
```

Clearly, the side-effects define the lengthy part of the semantics. Hence, before translating the IR into a Boolean formula for VSA, we perform liveness analysis [26] and in order to eliminate redundant assignments, which do not have any effect on the program execution. This technique typically simplifies the programs — and thus the resulting Boolean formulae — significantly because most side-effects do not influence any further program execution, and so does liveness analysis for the given example:

```
0x42 : AND.B R1 R1 #15;
0x43 : ADD.B R0 R0 R1;
0x44 : MOV.b R95:0 R0:7; LSL.B R0 R0 #1;
0x45 : BRANCH (R95:0) label #0x46;
```

Indeed, similar reductions can be observed for all our benchmark programs. It is thus meaningful with respect to tractability to decouple the explicit effects of an instruction from its side-effects.

3.2 Bit-Blasting Blocks

Expressing the semantics of a block in Boolean logic has become a standard technique in program analysis due to the rise in popularity of SAT-based bounded model checkers [11]. To provide a formula that describes the semantics of the simplified block, we first apply SSA conversion (which ensures that each variable is assigned exactly once). We then have bit-vectors $\boldsymbol{V} = \{\boldsymbol{r0}, \boldsymbol{r1}\}$ on input of the block, bit-vectors $\boldsymbol{V'} = \{\boldsymbol{r0'}, \boldsymbol{r1'}, \boldsymbol{r95'}\}$ on output, and an additional intermediate bit-vector $\boldsymbol{r0}''$. The most sophisticated encoding is that of the `ADD` instruction, which is encoded as a full adder with intermediate carry-bits $\boldsymbol{c}$. Given these bit-vectors, the instructions are translated into Boolean formulae is follows:

$$\begin{aligned}
\varphi_{\texttt{0x42}} &= \textstyle\bigwedge_{i=0}^{3}(\boldsymbol{r1}'[i] \leftrightarrow \boldsymbol{r1}[i]) \wedge \bigwedge_{i=4}^{7}(\neg \boldsymbol{r1}'[i]) \\
\varphi_{\texttt{0x43}} &= \begin{array}{l} (\bigwedge_{i=0}^{7} \boldsymbol{r0}''[i] \leftrightarrow \boldsymbol{r0}[i] \oplus \boldsymbol{r1}'[i] \oplus \boldsymbol{c}[i]) \wedge \neg \boldsymbol{c}[0] \wedge \\ (\bigwedge_{i=0}^{6} \boldsymbol{c}[i+1] \leftrightarrow (\boldsymbol{r0}[i] \wedge \boldsymbol{r1}'[i]) \vee (\boldsymbol{r0}[i] \wedge \boldsymbol{c}[i]) \vee (\boldsymbol{r1}'[i] \wedge \boldsymbol{c}[i])) \end{array} \\
\varphi_{\texttt{0x44}} &= (\boldsymbol{r95}'[0] \leftrightarrow \boldsymbol{r0}''[7]) \wedge (\textstyle\bigwedge_{i=1}^{7} \boldsymbol{r0}'[i] \leftrightarrow \boldsymbol{r0}''[i-1]) \wedge \neg \boldsymbol{r0}'[0]
\end{aligned}$$

Observe that instruction `0x45` does not alter any data, and is thus not included in the above enumeration. Then, the conjoined formula

$$\varphi = \varphi_{\texttt{0x42}} \wedge \varphi_{\texttt{0x43}} \wedge \varphi_{\texttt{0x44}}$$

describes how the block relates the inputs $\boldsymbol{V}$ to the outputs $\boldsymbol{V}'$ using some intermediate variables (which are existentially quantified). In the remainder of this example, we additionally assume that our analysis framework has inferred that `R0` is in the range of 110 to 120 on input of the program, and that φ has been extended with this constraint.

3.3 Value-Set Analysis for Extended Bit-Vectors

The algorithm of Barrett and King [4, Fig. 3] computes the VSA of a bit-vector $\boldsymbol{v}$ in unsigned or two's complement representation as constraint by some Boolean formula ψ. It does so by converging onto the value-sets of $\boldsymbol{v}$ using over- and under-approximation. However, the drawback of their method is that it requires $\mathit{vars}(\psi) = \boldsymbol{v}$, i.e., ψ ranges only over the propositional variables in $\boldsymbol{v}$. To apply the method to the above formula φ and compute the value-sets of $\boldsymbol{r0} \in \boldsymbol{V}$ on entry, e.g., it is thus necessary to eliminate all variables $\mathit{vars}(\varphi) \setminus \boldsymbol{r0}$ from φ using existential quantifier elimination. Intuitively, this step removes all information pertaining to the variables $\mathit{vars}(\varphi) \setminus \boldsymbol{r0}$ from φ. In what follows, denote the operation of projecting a Boolean formula ψ onto a bit-vector $\boldsymbol{v} \subseteq \mathit{vars}(\psi)$ by $\pi_{\boldsymbol{v}}(\psi)$. In our framework, we apply the SAT-based quantifier elimination scheme by Brauer et al. [8], though other approaches [22] are equally applicable.

Projecting onto Extended Bit-Vectors As stated before, it is our desire to reason about the values of register `R0` on the entries of both successors of instruction `0x45`. These values correspond to the values of the bit-vector $\boldsymbol{r0}'$. Yet, we also need to take into account the relationship between $\boldsymbol{r0}'$ and the carry flag $\boldsymbol{r95}'[0]$. We therefore treat $\boldsymbol{o} = \boldsymbol{r0}' : \boldsymbol{r95}'[0]$, where : denotes concatenation, as the target bit-vector for VSA, and project φ onto $\boldsymbol{o}$. Then, $\pi_{\boldsymbol{o}}(\varphi)$ describes a Boolean relationship between $\boldsymbol{r0}'$ and the carry-flag $\boldsymbol{r95}'[0]$.

Value-Set Analysis We finally apply the VSA to $\pi_{\boldsymbol{o}}(\varphi)$ so as to compute the unsigned values of `R0` on entry of both successor blocks of `0x45`. To express the unsigned value of a bit-vector $\boldsymbol{v} = \langle \boldsymbol{v}[0], \ldots, \boldsymbol{v}[n] \rangle$, let $\langle\!\langle \boldsymbol{v} \rangle\!\rangle = \sum_{i=0}^{n-1} 2^i \cdot \boldsymbol{v}[i]$. Since `R0` is an 8-bit register, and the representing bit-vector is extended by the carry-flag to give $\boldsymbol{o}$, we clearly have $0 \leq \langle\!\langle \boldsymbol{o} \rangle\!\rangle \leq 2^9 - 1$. Applying VSA then yields the following value-sets:

$$\begin{aligned} \langle\!\langle \boldsymbol{o} \rangle\!\rangle \in \{\ & 220, 222, \ldots, 252, 254, \\ & 256, 258, \ldots, 268, 270\} \end{aligned}$$

Observe that the values in the range $2^8 \leq \langle\!\langle \boldsymbol{o} \rangle\!\rangle \leq 2^9 - 1$ reduced by 2^8 correspond to those values for which the branch is taken. Likewise, the values of $\langle\!\langle \boldsymbol{o} \rangle\!\rangle$ in the range $0 \leq \langle\!\langle \boldsymbol{o} \rangle\!\rangle \leq 2^8 - 1$ correspond to the values for which the branch is not taken. Hence, the results of VSA can be interpreted as follows:

1. The value-set $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \in \{220, 222, \ldots, 252, 254\}$ is propagated into the successor block `0x46`. This is because it is possible that the branch is not taken for these values.
2. The value-set $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \in \{256, 258, \ldots, 268, 270\}$ is reduced by 256 so as to eliminate the set carry-flag, which gives $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \in \{0, 2, \ldots, 12, 14\}$ as potential values if the branch is taken.

In this example, the definition of the carry-flag is straightforward: the most significant bit of `R0` in instruction `0x44` is moved into the carry. This is clearly not always the case. As an example, recall the lengthy definition of the effects of `ADD R0 R1` on the carry-flag in Sect. 2.1 (consisting of one negation, three conjunctions and three disjunctions). By encoding these relations in a single formula and projecting onto the carry-flag conjoined with the target register, our analysis makes such relations explicit.

4 Weak Relations Between Registers

It is interesting to observe that the approach can likewise be applied to derive relations between different bit-vectors which, in turn, represent different registers. Suppose we apply the same strategy to the extended bit-vector $\boldsymbol{o}' = \boldsymbol{r0}' : \boldsymbol{r0}[7]$. Applying VSA to $\boldsymbol{o}'$ then yields results in the range $0 \leq \langle\!\langle \boldsymbol{o}' \rangle\!\rangle \leq 2^9 - 1$. Following from the encoding of unsigned integer values, the results exhibit which values $\boldsymbol{r0}'$ can take for inputs such that either $0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 127$ or $128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 255$. If VSA yields a value such that the most significant bit of $\boldsymbol{o}'$ is set, then $\langle \boldsymbol{o}'[0], \ldots, \boldsymbol{o}'[7] \rangle$ is a value which is reachable if $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \geq 128$.

However, a more precise characterization of the relation between $\boldsymbol{r0}$ and $\boldsymbol{r0}'$ can be obtained by applying VSA to $\boldsymbol{o}'' = \boldsymbol{o}' : \boldsymbol{r0}[6]$, which partitions the values according to the inputs (i) $0 \leq 63$, (ii) $64 \leq 127$, (iii) $128 \leq 191$, and (iv) $192 \leq 255$. Yet, the payoff for the increase in expressiveness is higher computational complexity. In fact, the payoff is two-fold. First, the efficiency of SAT-based quantifier elimination decreases as the number of propositional variables to project onto increases. Second, the size of the resulting value-sets increases, and thus the number of SAT calls to compute them.

5 Experimental Evidence

We have implemented the techniques discussed in this paper in JAVA using the SAT4J solver [23]. The experiments were performed with the expressed aim of answering the following questions:

- How does the translation of the instructions into an IR affect the performance of SAT-based value-set analysis? This is of interest since the decoupling of the side-effects from the *intended* effect of the instruction allows for a more effective liveness analysis than implemented in tools such as [MC]SQUARE [31].
- How does analyzing extended bit-vectors affect the overall performance compared to the SAT-based analysis discussed in [29]. Their analysis recovers weakly-relational information using alternating executions of forward and backward analysis so as to capture the relation between, e.g., a register `R0` and the carry-flag after a branching has been analyzed, whereas our analysis tracks such information beforehand.

We have applied the analysis to various benchmark programs for the INTEL MCS-51 microcontroller, which we have used before to evaluate the effectiveness of our analyses [29, Sect. 4]. VSA is used to compute the target addresses of indirect jumps, where bit-vectors are extended based on the status flags that trigger conditional branching (like the carry-flag in the worked example). Decoupling the instructions from the side-effects led to a reduction in size of the Boolean formulae of at least 75%. Experimental results with respect to runtime requirements are shown in Tab. 1. Compared to the analysis in [29], the runtime decreases by at least 50% for the benchmarks, due to fewer VSA executions. The computed value-sets are identical for this benchmark set.

Table 1. Experimental results for SAT-based VSA

Name	LoC	# instr.	Runtime
Single Row Input	80	67	1.42s
Keypad	113	113	1.93s
Communication Link	111	164	1.49s
Task Scheduler	81	105	6.77s
Switch Case	82	166	8.09s
Emergency Stop	138	150	0.91s

To investigate the portability of our IR to other architectures, we have implemented a compiler from ARM assembly to the sketched IR. We have done so within the METAMOC [17] toolchain which already provides support for disassembling ARM binaries and reconstructing some control flow. Furthermore, METAMOC contains formal descriptions of instruction effects. We translated these formal descriptions to the required IR format manually, requiring approximately one day. Translating a different platform to the IR uncovered a few areas where we might beneficially extend our intermediate language: the ARM architecture excessively uses conditional execution of instructions. In this situation, an instruction is executed if some logical combination of bits evaluates to *true*; otherwise, the instruction is simply skipped. Compilers for ARM use such constructs frequently to simplify the control structure of programs, leading to fewer branches. Adding support for such instruction features is fundamental to support different hardware platforms. We have chosen to support such behavior by means of *guarded execution*. Each operation can be annotated with a guard. If the guard evaluates to *true*, the corresponding instruction is executed, and otherwise it is the identity. The translation of this construct into Boolean logic is then trivial.

6 Related Work

In abstract interpretation [14], even for a fixed abstract domain, there are typically many different ways of designing the abstract operations. Ideally, the abstract operations should be as descriptive as possible, although there is usually interplay with accuracy and complexity. A case in point is given by the seminal work of Cousot and Halbwachs [15, Sect. 4.2.1] on polyhedral analysis, which discusses different ways of modeling multiplication. However, designing transfer functions manually is difficult (cp. the critique of Granger [18] on the difficulty of designing transformers for congruences), there has thus been increasing interest in computing the abstract operations from their concrete versions automatically, as part of the analysis itself [5,6,20,21,24,27,28,30]. In their seminal work, Reps et al. [30] showed that a theorem prover can be invoked to compute an transformer on-the-fly, during the analysis, and showed that their algorithm is feasible for any domain that satisfies the ascending chain condition. Their approach was later put forward for bit-wise linear congruences [21] and affine relations [5]. Both approaches replace the theorem prover from [30] by a SAT solver and describe the concrete (relational) semantics of a program (over finite bit-vectors) in propositional Boolean logic. Further, they abstract the Boolean formulae offline and describe input-output relations in a fixed abstract domain. Although the analysis discussed in this paper is based on a similar Boolean encoding, it does not compute any transformers, but rather invokes a SAT solver dynamically, during the analysis. Contemporaneously to Reps et al. [30], it was observed by Regehr et al. [27,28] that BDDs can be used to compute best transformers for intervals using interval subdivision. The lack of abstraction in their approach entails that the runtimes of their method are often in excess of 24h, even for 8-bit architectures.

The key algorithms used in our framework have been discussed before, though in different variations. In particular, VSA heavily depends on the algorithm in [4, Fig. 3], which is combined with a recent SAT-based projection scheme by Brauer et al. [8]. Comparable projection algorithms have been proposed before [22,25], but they depend on BDDs to obtain a CNF representation of the quantifier-free formula (which can be passed to the SAT solver for value-set abstraction). By way of comparison, using the algorithm from [8] allows for a lightweight implementation. The value-set abstraction, in turn, extends an interval abstraction scheme for Boolean formulae using a form of dichotomic search, which has (to the best of our knowledge) first been discussed by Codish et al. [12] in the context of logic programming. Their scheme has later been applied in different settings, e.g., in transfer function synthesis [6] or a reduced product operator for intervals and congruences over finite integer arithmetic [7]. Reinbacher and Brauer [29] have proposed a similar technique for control flow recovery from executable code, but they do not extend their bit-vectors for VSA. They thus combine SAT-based forward analysis with depth-bounded backward analysis to propagate values only into the desired successor branches.

Over recent years, many different tools for binary code analysis have been proposed, the most prominent of which probably is CodeSurfer/x86 [2]. Yet, since the degree of error propagation is comparatively high in binary code analysis (cp. [28]), we have decided to synthesize transfer functions (or abstractions, respectively) in our tool [mc]square [31] so as to keep the loss of information at a minimum.

7 Concluding Discussion

In essence, this paper advocates two techniques for binary code analysis. First of all, it argues that SAT solving provides an effective as well as efficient tool for VSA of bit-vector programs. Different recent algorithms — most notably projection using prime implicants and dichotomic search — are paired to achieve this. The approach thereby benefits from the progress on state-of-the-art SAT solvers. Secondly, the efforts required to implement a SAT-based program analysis framework largely depend on the complexity of the target instruction set. To mitigate this problem, we have proposed an intermediate representation based on decomposing instructions and their side-effects into sequences of basic operations. This significantly eases the implementation efforts and allows us to port our framework to different hardware platforms in a very short time frame. Our experiences with the AVR ATmega, Intel MCS-51 and ARM9 hardware platforms indicates that adding support for a hardware platform can easily be achieved within one week, whereas several man-months were required otherwise. In particular, testing and debugging the implementation of the Boolean encodings is eased. In this paper, we have not presented a formal semantics for the IR, mostly because it is straightforward to derive such a semantics from existing relational semantics for flow-chart programs over finite bit-vectors. Examples of such semantics are discussed in [21, Sect. 4] or [13, Sect. 2.1].

Acknowledgements Jörg Brauer and Stefan Kowalewski were supported, in part, by the DFG research training group 1298 Algorithmic Synthesis of Reactive and Discrete-Continuous Systems and the by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication, German Research Foundation grant DFG EXC 89. The authors want to thank Axel Simon for interesting discussions about the design of intermediate representations.

References

1. Atmel Corporation. *8-bit AVR Instruction Set*, July 2008.
2. G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S.-H. Yong, C. H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 158–163. Springer, 2005.
3. P. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In *CAV*, 2011. To appear.
4. E. Barrett and A. King. Range and Set Abstraction Using SAT. *Electronic Notes in Theoretical Computer Science*, 267(1):17–27, 2010.
5. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.
6. J. Brauer and A. King. Transfer Function Synthesis without Quantifier Elimination. In *ESOP*, volume 6602 of *LNCS*, pages 97–115. Springer, 2011.
7. J. Brauer, A. King, and S. Kowalewski. Range analysis of microcontroller code using bit-level congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.
8. J. Brauer, A. King, and J. Kriener. Existential quantification as incremental SAT. In *CAV*, volume 6806 of *LNCS*, pages 191–208. Springer, 2011.
9. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, 2011. To appear.
10. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98:142–170, 1992.
11. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
12. M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.
13. B. Cook, D. Kroening, P. Rümmer, and C. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *TACAS*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.
14. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.
15. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.
16. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Effciently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, pages 451–590, 1991.
17. A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *WCET*, pages 113–123, 2010.
18. P. Granger. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *TAPSOFT 1991*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.
19. M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.
20. A. King and H. Søndergaard. Inferring congruence equations using SAT. In *CAV*, volume 5123 of *LNCS*, pages 281–293. Springer, 2008.
21. A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.

22. S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.
23. D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. `http://www.sat4j.org/`.
24. D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.
25. D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
26. R. Muth, S. K. Debray, S. A. Watterson, and K. De Bosschere. Alto: A Link-Time Optimizer for the Compaq Alpha. *Softw., Pract. Exper.*, 31(1):67–101, 2001.
27. J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES*, pages 34–43. ACM, 2006.
28. J. Regehr and A. Reid. HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.
29. T. Reinbacher and J. Brauer. Precise Control Flow Recovery Using Boolean Logic. In *EMSOFT*, 2011. To appear.
30. T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
31. B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–27, 2010.

Automatic Derivation of Abstract Semantics From Instruction Set Descriptions

Dominique Gückel and Stefan Kowalewski

Embedded Software Laboratory
RWTH Aachen University Ahornstr. 55
Aachen, Germany
`<gueckel, kowalewski>@embedded.rwth-aachen.de`

Abstract. Abstracted semantics of instructions of processor-based architectures are an invaluable asset for several formal verification techniques, such as software model checking and static analysis. In the field of model checking, abstract versions of instructions can help counter the state explosion problem, for instance by replacing explicit values by symbolic representations of sets of values. Similar to this, static analyses often operate on an abstract domain in order to reduce complexity, guarantee termination, or both. Hence, for a given microcontroller, the task at hand is to find such abstractions. Due to the large number of available microcontrollers, some of which are even created for specific applications, it is impracticable to rely on human developers to perform this step. Therefore, we propose a technique that starts from imperative descriptions of instructions, which allows to automate most of the process.

1 Introduction

Formal verification of software for embedded systems is crucial for multiple reasons. First of all, such systems are often used in safety-critical fields of applications, such as chemical plants, where failures of the controlling system may result in severe injuries or even fatalities. Furthermore, applying corrections after delivering a system to the customer may be inpracticable or costly, for instance in the case of devices embedded into cars. Such scenarios may be avoided by formal verification, for instance software model checking [4, 2].

1.1 Focus

The focus of our work is model checking and static analysis [5] of binary code for microcontrollers. For this purpose, we need to lift the given concrete semantics of the instructions of which the binary consists to their abstract counterparts in the respective domain. In the case of model checking, the sought-after abstract version of each instruction should be able to operate not only on conventional two-valued boolean logic, but on a variant of three-valued logic. This allows for certain abstractions to be applied, which can help avoid the state explosion problem. In the case of static analysis, the abstracted instruction should provide information on memory locations it reads and writes, plus on how executing it affects the control flow.

Deriving abstract semantics from concrete semantics is a task that is usually performed manually, thus exploiting the knowledge of an expert in the associated fields. While this may be suitable for verification tools that are not likely to be modified very often, such as for high level languages, this is not applicable in the embedded domain, where a wide variety of different platforms is available to the developer. In case the platform is switched to a different microcontroller, which uses a different instruction set, the previous work on abstraction has to be done anew.

1.2 Approach

In order to reduce the necessary effort, we propose to conduct the abstraction on an already executable form of the instructions, that is, a description in an imperative programming language. In our setting, such a description can also be used to generate an instruction set simulator, which can build the state space for model checking programs for the described platform.

1.3 Contribution

In this paper, we make the following contributions:

– We describe how microcontroller instruction sets can be translated into a form that allows for automatic analysis of certain properties.
– Based on the results of these analyses, we can then derive abstract semantics that are more suitable for state space building than the concrete semantics. As an example, we detail how to obtain the necessary semantics for an abstraction called *delayed nondeterminism*, which can be used in model checking.
– We detail the generation of static analyzers for different platforms based on the aforementioned analyses.

1.4 Outline

The rest of this paper is structured as follows. In Sect. 2, we illustrate the tools we used in our contribution. Next, to motivate our work, we provide an example that illustrates the effects of an inappropriate modeling of instructions. The actual work on automatically deriving abstract semantics is contained in Sect. 4. The next-to last section focuses on related work (Sect.5), and Sect. 6 concludes this paper.

2 Preliminaries

In this section, we detail the environment of our contribution. First, we summarize the main features of the [MC]SQUARE model checker, which uses several of the abstraction techniques we are interested in. Next, we focus on a specific technique, called delayed nondeterminism. Finally, we present some features of a hardware description language that serves as a starting point for our automatic derivation of instruction semantics.

2.1 The [mc]square Assembly Code Model Checker

[mc]square [13] is an explicit-state model checker for microcontroller binary code. Given a binary program and a formula in Computation Tree Logic (CTL), [mc]square can automatically verify whether the program satisfies the formula, or create a counterexample in case the program violates the formula. Atomic propositions in formulas may be statements about the values of general purpose registers, I/O registers, and the main memory. Currently, [mc]square supports the Atmel ATmega16 and ATmega128, Intel MCS-51, and Renesas R8C/23 microcontrollers. Furthermore, it can verify programs for Programmable Logic Controllers (PLCs) written in Instruction List (IL).

[mc]square builds state spaces for conducting the actual model checking by means of special simulators. These can execute the programs under consideration just like simulators provided by hardware vendors, by applying the semantics of instructions to a model of the system's memories, and simulating the effects of interrupts, I/O ports, and on-chip peripherals. The key difference, however, is that simulators in [mc]square support nondeterminism to model unknown values, and also provide certain abstractions. Nondeterminism is introduced into the system by I/O ports, timers, and interrupts. I/O ports communicate with the environment, of which we have to assume that it can show any behavior, i.e., any value might be present in I/O registers. Timers are modeled using nondeterminism because [mc]square deliberately abstracts from time, resulting in the value of timer registers to be unknown. Finally, interrupts are nondeterministic events because an active interrupt may occur or not occur, and both cases need to be considered for model checking. In case a nondeterministic bit has to be *instantiated* to a deterministic 0 or 1, the simulator performs the necessary step.

The state creation process in [mc]square operates as follows:

– Load a state into the simulator.
– Determine assignments needed for resolving nondeterminism.
– For each assignment
  • If the assignment indicates the occurrence of an enabled interrupt, simulate the effect of that interrupt. Otherwise, execute the current instruction.
  • Evaluate truth values of atomic propositions.
– Return resulting states.

Using and resolving nondeterminism creates an over-approximation of the behavior exhibited by the real hardware, allowing [mc]square to check for safety properties. Instantiation of n nondeterministic bits usually results in 2^n successor states (i.e., exponential complexity), which is why immediate instantiation of all nondeterministic bits is infeasible. Therefore, several abstraction techniques are implemented in [MC]SQUARE to prevent this. Within the scope of this paper, we focus on two of these: first of all, a technique called *delayed nondeterminism*, details on which are given in the next section, and second, on techniques that are enabled by static analyses. The latter is an optional preprocessing step performed before conducting the actual model checking, during which analysis results can be used to apply abstractions such as Dead Variable Reduction [17, 14].

2.2 Delayed Nondeterminism

An instantiation of nondeterministic bits results in an exponential number of successor states. Deterministic simulation triggers instantiation whenever an instruction accesses a nondeterministic memory cell, hence it cannot avoid the exponential blowup. However, at least on RISC-like load-store-architectures like the Atmel ATmega microcontrollers, the instruction in question usually only *copies* the content of the cell to some other cell, for instance a register. It does not modify the value or use it as an argument, as would an arithmetic or logic instruction. *Delayed nondeterminism* [11] is an abstraction exploiting this observation. Instead of immediately resolving the nondeterminism, a nondeterministic value is *propagated* through memory. Only when an instruction actually needs the deterministic value, the cell is instantiated. As a result, all the paths starting at the original read instruction are not created at all. This approach proves particularly useful in case a value consisting of multiple nondeterministic bits is read, of which only a few bits are actually needed, for instance by an instruction testing a single bit.

2.3 Description of Microcontrollers Using SGDL

The concept of [MC]SQUARE requires the tool to be hardware-dependent. While this provides great accuracy as to hardware peculiarities, and also the ability to provide easy to understand counterexamples, it necessarily results in the obvious disadvantage of additional effort whenever adding support for a new platform. In order to compensate for this, [MC]SQUARE features an extensible architecture, and additionally contains a complete programming system for creating simulators in a high level language. The language is called SGDL, and a compiler for SGDL is part of [MC]SQUARE. SGDL is a hardware description language specifically tailored to describe microcontroller architectures, providing elements for describing entities such as instructions, memories, and interrupts. In the following, we only introduce those parts relevant for analyzing instruction semantics. Further details on SGDL are provided in [7, 6], and details on its precursor language from the AVRora project are available from [16].

Example 1. Excerpt from the SGDL description of the Intel MCS-51

```
format OPCODE_IMM8_IMM8 = {opcode[7:0], imm8_1[7:0], imm8_2[7:0]};

subroutine performCJNE(leftVal:ubyte, rightVal:ubyte,
   target:SIGNED_IMM8) : void {
   if (leftVal != rightVal) {
      $pc = $pc + target;
      if (leftVal < rightVal) $CY = true;
      else $CY = false;
   }
};

instruction "cjne_acc_direct_rel" {
   encoding = OPCODE_IMM8_IMM8 where {opcode = 0b10110101};
   operandtypes = {imm8_1 : IMM8, imm8_2 : SIGNED_IMM8};
   instantiate = {};
   dnd instantiate = {};
   execute = {
      performCJNE($ACC, $sram(imm8_1), imm8_2);
   };
};
```

In the example, an instruction called `cjne_acc_direct_rel` is declared. The binary encoding of this instruction consists of an 8 bit wide opcode and two operands, each of which is also 8 bits wide. Within the scope of this instruction, these 8 bit operands are to be interpreted as signed 8 bit integers, using two's complement representation. The concrete semantics are described within the `execute` section of the `instruction` element. Global variables, i.e., the resource model of the simulated device, are accessed by prefixing the according identifier with a $ or a # (not in this example), whereas local variables are always accessed with neither. Function calls are also possible, in this example for externalizing the CJNE functionality, which is shared by the different variants of the CJNE instruction (the MCS-51 instruction set contains four of these, each for a different addressing mode).

In case an instruction may encounter nondeterministic values in some addresses it accesses, the developer can indicate that the simulator should instantiate these by adding an `instantiate` entry to the instruction. Any address contained in the set will be instantiated. The `dnd instantiate` section has the same semantics, but is used only when the simulation type is set to *use delayed nondeterminism.*

Global memories in SGDL consist of two parallel structures to allow for nondeterminism. The first structure is the *value*, which is accessed using the aforementioned dollar symbol. The second structure is the *nondeterminism mask*. Both structures together represent values in ternary logics, with the semantics that a bit is nondeterministic iff its nondeterminism mask is set to 1. If the mask bit is set to 1, then the content of *value* becomes irrelevant, as logically, it could be either 0 or 1. Hence, generated simulators force it to 0, thus guaranteeing consistent states and additionally removing a potential distinguishing feature of states (which in some cases reduces the size of the state space).

A typical instruction set description in SGDL contains between 2.000 and 4.000 lines of code, depending on the number of instructions and overall complexity of the device.

2.4 Notations

Definition 1. ***Alphabet for ternary logics***
The alphabet for ternary logics is defined as $\Sigma := \{0, 1, n\}$. A word of length m over Σ^* is then a sequence of letters representing bits that are either explicitly 0, 1, or could be both.

Example 2. The word $w := 000n\,0000$ can be instantiated to the explicit values 0000 0000 and 0001 0000.

Definition 2. ***Values of a memory cell***
Let x be a memory cell of m bits width. Then

- $\texttt{val}(x)$ *denotes the content of x.*
- $\texttt{ndm}(x)$ *denotes the content of the nondeterminism mask of x.*

`val` *and* `ndm` *are bit vectors that can be combined to represent a value in ternary logic. Whenever a bit in* $\texttt{ndm}(x)$ *is set to 1, then that bit is considered to be $n \in \Sigma$, i.e., the content of* $\texttt{val}(x)$ *for that bit becomes irrelevant.*

3 A Motivational Example

As an example, consider the following instruction, which is part of the instruction set of the Atmel AVR family of microcontrollers:

```
IN R0, TIFR
```

This instruction reads the value of the *timer interrupt flag register*, `TIFR`, and copies it into the general purpose register (GPR) R0. No flags are altered by this instruction. Accordingly, the semantics of the instruction, as depicted in the instruction set manual (ISM) are

$$\texttt{Rd} \leftarrow \texttt{I/O}$$

where `Rd` is a GPR, and `I/O` is an I/O register. Being an I/O register, `TIFR` may contain nondeterministic data. Hence, we either need to instantiate all nondeterministic bits immediately, or propagate this information into the destination, in this example `R0`. For simulation (i.e., state space building), the optimal abstract semantics would be

$$\texttt{val(R0)} \leftarrow \texttt{val(TIFR)} \tag{1}$$
$$\texttt{ndm(R0)} \leftarrow \texttt{ndm(TIFR)} \tag{2}$$

To achieve the latter, we have several options:

1. Implement explicit code for copying. This approach requires a human developer to inspect the instruction semantics in the ISM, and implement the necessary code into the simulator.
2. Naive automatic approximation. Whenever an instruction depends on at least one nondeterministic input bit
   – Check if all output bits allow nondeterminism. To obtain the output bits without analyzing the instruction code, it suffices to execute the instruction once, then revert the resulting machine state to the original state.
   – If all output bits allow nondeterminism, set all of them to nondeterministic.
   – Else instantiate all input bits, and execute the instruction using the concrete semantics from the ISM.

While the first approach, using a developer, can always yield the optimal solution, it is also the most inappropriate one. Instruction sets usually consist of hundreds of instructions, and each of those has to be lifted from the concrete to the abstract. Furthermore, this is a very simple example, in which the developer can hardly introduce any mistakes. Other examples, like complex arithmetic instructions, can easily cause the developer to forget maybe the one or other flag bit, which may result in the state space generator containing a correct implementation for one simulation type (e.g., fully deterministic simulation based on the concrete semantics), and a faulty one for another type (e.g., delayed nondeterminism).

Compared to this, the proposed naive implementation of the automatic approach certainly has the advantage of far less manual effort. Moreover, it guarantees an over-approximation of instruction behavior, therefore preserving the model checker's ability to check safety properties. The disadvantage, however, is that it is grossly inaccurate. Consider the following machine state:

$$\begin{aligned} \texttt{val(R0)} &= 0000\,0001 && (3)\\ \texttt{ndm(R0)} &= 0000\,0000 && (4)\\ \texttt{val(TIFR)} &= 0000\,0000 && (5)\\ \texttt{ndm(TIFR)} &= 1000\,0000 && (6) \end{aligned}$$

Executing the example instruction using the naive abstract semantics will change this to the following machine state:

$$\begin{aligned} \texttt{val(R0)} &= 0000\,0000 && (7)\\ \texttt{ndm(R0)} &= 1111\,1111 && (8)\\ \texttt{val(TIFR)} &= 0000\,0000 && (9)\\ \texttt{ndm(TIFR)} &= 1000\,0000 && (10) \end{aligned}$$

That is, the source register, `TIFR`, retains its single nondeterministic bit, while the previously deterministic target register `R0` becomes completely nondeterministic. The consequences of this change depend entirely on the next instructions accessing

`R0` (note that this need not necessarily be the immediately next instructions). In case the next instruction accessing `R0` is a bit test instruction such as `SBIC` (i.e., skip next instruction if bit is clear), only a single bit may be instantiated. In this case, the naive approach would yield the desired result, which is to avoid instantiation until the value of nondeterministic bits is actually needed. However, in case the next instruction is a comparison, such as `CPI R0, 128`, the naive approach will result in an instantiation of 8 nondeterministic bits, yielding 256 successor states. Compared to this, the optimal approach would copy only 1 nondeterministic bit from TIFR to R0, thus the instantiation triggered by executing `CPI` would result in only 2 successors.

The disadvantage of the naive implementation is due to the fact that it marks all bits written by the instruction as nondeterministic. Such an approximation is overly pessimistic for `IN`, as there is a direct mapping of input bit i to output bit i in the equally wide registers `TIFR` and `R0`. For operations such as `ADDC Rd, Rr` (addition with carry), however, there is no such mapping. Instead, the value of a target bit may depend on the values of *several* bits in the input. Thus, without any additional knowledge about the actions performed by an instruction, the pessimistic assumption that any output may result, is actually an appropriate one. In the following sections, we illustrate a concept how to gain such information, and produce a smaller over-approximation.

4 Deriving Abstract Semantics

In this section, we focus on abstract semantics for delayed nondeterminism and static analysis. Throughout the section, we use the term *input* of an instruction as a synonym for the sets of locations read by it, and analogously the term *output* for the set of written locations.

4.1 Prerequisites

Certain invariants regarding memory locations must always hold in both the concrete and the abstract semantics of instructions, as they are needed to preserve expressiveness:

– Operands. We assume that operands are always deterministic. This is guaranteed by construction, as they are instantiated once by the disassembler.
– Addresses. Addresses must always be deterministic, as a nondeterministic address used in an instruction may result in any (visible) address to be read or written. Especially on devices with memory-mapped I/O, this could also have an impact on device behavior.
– Control flow. Any memory location relevant for control flow must remain deterministic. This applies to status registers, but not to general purpose registers. Nondeterministic control flow is undesirable because of a severely reduced expressiveness (e.g., a status register indicating that the last computation yielded a result that was zero, negative, and odd). The direct implication

is that control-flow relevant instructions must always operate on deterministic data.
- Arithmetics and logics. Any computation involving a nondeterministic value yields a nondeterministic result. If the target of the assignment requires the result to be deterministic, then all variables involved in the computation have to be instantiated first.

4.2 Identifying the Control Flow Type

As pointed out in Sect. 4.1, all control-flow relevant operations require their input to be deterministic. The task at hand is therefore to separate jump and branch instructions from arithmetic/logic and data transfer instructions. Furthermore, for our second goal, generating an operative static analyzer, we do not only need to generate transfer functions for each instruction (i.e., an abstract semantics), but also establish a flow relation for creating the control flow graph (CFG). The latter requires analyses to find out the number of succeeding instructions, and their addresses.

Both goals can be achieved by means of certain static analyses of the `execute` sections and subroutines in the SGDL description. As the language supports function calls, all analyses have to be conducted interprocedurally. Three analyses suffice: Reaching Definitions Analysis (RDA), Read Variable Analysis (RVA), and Written Variable Analysis (WVA). RDA is a standard textbook analysis [10], RVA is basically the collecting semantics of Live Variables Analysis (LVA) (i.e., an LVA with a constant empty *kill* function), and WVA is the counterpart of RVA with respect to written variables. The overall idea is to analyze write accesses to the program counter. Figure 1 illustrates the classification algorithm, which works as follows:

- Construct the control flow graphs for the current instruction and all called functions.
- All instructions implicitly increment the program counter by their own size, so insert one reaching definition (RD) into the entry node of the instruction CFG.
- Conduct the analyses.
- Classify instructions based on the number and origin of RDs reaching the exit node:
  - 1 RD from the entry node: regular instruction
  - 1 RD, but not from the entry node: program counter is inevitably overwritten with a single value, i.e., an unconditional *jump* instruction
  - 2 RDs: a conditional *jump*
- Refine the classification: *jump* instructions manipulating the stack could be *call* or *return* instructions, depending on whether they read / write the content of the program counter from / to the stack. Use RVA and WVA results to distinguish these.

The second step, obtaining the value written at runtime into the program counter, is part of the next section. Technically, it consists of two steps: first, use

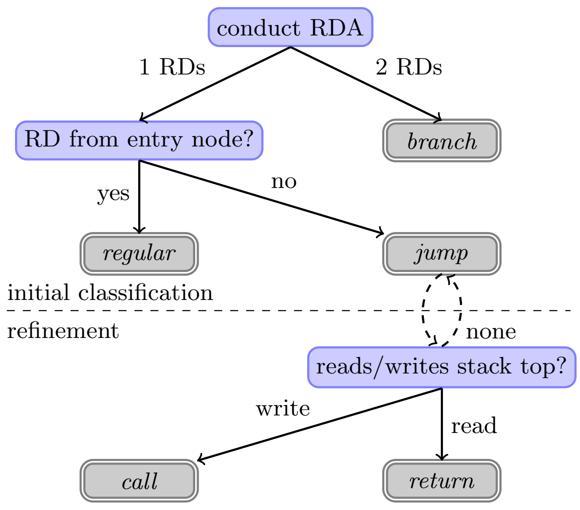

Fig. 1. Classification strategy for control flow type

the RDA to locate assignments to the program counter, and second, backtrack and collect all subexpressions on the right hand side of such assignments. The resulting expression can then be evaluated at runtime on concrete instances of the instruction.

4.3 Analysis of Data Flow

An analysis of the data flow has to identify the effects of individual assignments to global variables. The goal of the analysis is to identify possible propagations of nondeterminism and also the opposite, variables that must not be nondeterministic when executing the instruction. Formally:

Let α be an instruction consisting of individual statements $\alpha_0, \ldots, \alpha_m$,

$$\begin{aligned}\alpha_i \in \{&\texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i}) \leftarrow \texttt{expr}_i,\\ &\texttt{localvar\_identifier} \leftarrow \texttt{expr}_i,\\ &\texttt{call}\,\texttt{f}_\texttt{i}(\texttt{args}_\texttt{i})\},\\ &1 \leq i \leq m \end{aligned} \quad (11)$$

Let $Addr_\alpha$ be the set of identifiers known to be used as an address within the scope of α, and initialize $Addr_\alpha := \emptyset$. Let $Var(expr)$ be the set of variables occurring in $expr$. Let $C \subseteq \mathbb{N} \times Addresses \times Addresses$ be a copy relation, wherein each entry is of the form $(instruction\ id, source, target)$.

Definition 3. ***Initial data flow analysis algorithm***
If α has been identified by the control flow algorithm as a jump, skip it. Else:
For all $\alpha_i \in \alpha$

– *If* $\alpha_i = \texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i}) \leftarrow \texttt{expr}_i$*:*
  - $Addr_\alpha := Addr_\alpha \cup \texttt{Var}(\texttt{addr\_expr}_\texttt{i})$
  - $Addr_\alpha := Addr_\alpha \cup \texttt{Var}(\texttt{addr})$ *for all addresses* `addr` *occuring in* $\texttt{expr}_i$*.*
  - *If* $\texttt{expr}_i = \texttt{memidentifier}_j(\texttt{addr})$ *for some memory identifier* j *and address* `addr`*, add an entry* $(i, \texttt{memidentifier}_j(\texttt{addr}),$ $\texttt{memidentifier}_\texttt{i}(\texttt{addr\_expr}_\texttt{i})$ *to the copy relation* C

– *If* $\alpha = \texttt{call}\,\texttt{f}_\texttt{i}(\texttt{args}_\texttt{i})\}$*: apply this algorithm to the called function to obtain a summary of function effects. Join resulting summary into analysis information of caller.*

Next, *refine* the initial analysis results by collecting subexpressions referenced in expressions. The goal is to relate identifiers used as addresses back to the operands and global resources visible at the beginning of the instructions's `execute` block. This can be achieved by a backwards search through the CFG. In case the analysis should fail in this for a given expression (possible due to branches in the CFG, indicating the value for a subexpression is not unique), there are two possible continuations, depending on the type of the expression: if the expression is known to be used an an address, either in reading or in writing, we need to mark all identifiers used in the instruction for instantiation. Otherwise, if the expression is known to be used as the right-hand side (rhs) value in an assignment, we have to replace it by a value that is marked *completely nondeterministic.*

4.4 Synthesis of Abstract Semantics

Following completion of control and data flow analysis, we can generate abstract semantics for each instruction.

Delayed Nondeterminism Instructions are considered as a list of individual assignments. For each of these, apply a translation rule. It is necessary to also add the original concrete semantics to the output because it might be necessary, during execution, to revert to it, and in the process of that, instantiate all nondeterminism in the input. This can happen if at least one of the assignments tries to write to an address that has to remain always deterministic (cf. the requirements detailed in Sect. 4.1).

Definition 4. ***Translation rules for assignments***
Let $\alpha_i = \texttt{memidentifier}(\texttt{addr\_expr}) \leftarrow \texttt{expr}$*.*

Let $\texttt{Inst}_\alpha$ *be the set of locations that have to be instantiated before executing* α*. Initially,* $\texttt{Inst}_\alpha := Var(\texttt{addr\_expr})$*.*

Replace $\texttt{addr\_expr}, \texttt{expr}$ *by the collected subexpressions, such that both expressions depend only on global variables and operands. If this is not possible because the analysis has failed, see below for a recovery strategy. Else, the abstract version* $\widehat{\alpha_i}$ *of* α *is defined by*

– *If the copy relation* C *created during analysis contains an entry* $(i, src, trgt)$*, then* $\widehat{\alpha_i} := [val(src) := val(trgt); ndm(src) := ndm(trgt)]$

– *Else (data is modified, so instantiate to generate concrete values)*
  - $\texttt{Inst}_\alpha := \mathit{Inst}_\alpha \cup \mathit{Var}(\texttt{expr})$
  - $\widehat{\alpha_i} := \alpha_i$

Recovery strategy: for expressions whose composition cannot be analyzed, the obvious solution is to assign a nondeterministic value to the target. In case this is not desirable, for instance because the target is a frequently accessed or very wide register (i.e., many nondeterministic bits would be created), a fallback would be to instantiate every input of this instruction, and use the concrete semantics instead. Thus, no improved semantics is available for this instruction, but at least it is guaranteed that the abstraction would not actually result in *state explosion instead of preventing it.*

Using these translation rules yields the semantics of delayed nondeterminism. An obvious improvement concerns the condition for instantiation, which, in the above version, is *if any computation is performed, then instantiate all inputs.* Therefore, all arithmetic instructions will instantiate all of their inputs because they necessarily contain at least one such α_i. A more permissive condition exploits the computation rules for ternary logic:

– For all operators in the input language, i.e., $+, -, *, /, \ldots$, introduce new abstract versions $\widehat{+}, \widehat{-}, \widehat{*}, \widehat{/}, \ldots$. Semantics are those of their concrete counterparts, except that the abstract versions operate also on nondeterministic (n) bits. For instance, $0 \mathbin{\widehat{+}} n = n \mathbin{\widehat{+}} 0 = 1 \mathbin{\widehat{+}} n = n \mathbin{\widehat{+}} 1 = n$, $0 \mathbin{\widehat{*}} n = 0, 1 \mathbin{\widehat{*}} n = n$, and analogously for all other operators.
– For each rhs expression in an assignment, create an abstract syntax tree representation
– Conduct a tree pattern matching, as described by Aho et al. [1], and apply tree rewriting rules, to replace the operators in the expression by their abstract counterparts.

These advanced rules then leave only two cases for forced instantiation of all inputs: first, an address expression that cannot be discomposed into its components, and second, an attempted write to a location marked as *must always remain deterministic.*

Static Analysis Using the results from the the control flow type analysis, it is possible to identify, for each instruction, the number of successors and their address, either absolute or relative. Therefore, given a program consisting of instances of these instructions, we can reconstruct the control flow graph from the disassembled binary representation of the program. Furthermore, the data flow analysis algorithm presented in the last section necessarily identifies read and written memory locations, i.e., provides a starting point for generating transfer functions for analyses such as RDA and LVA.

[MC]SQUARE already provides a framework for static analysis, which can conduct analyses in case the developer provides a CFG and transfer functions for the named analyses. Therefore, the actual generation of an operative analyzer is reduced to the task of generating the necessary code from the existing analysis results.

5 Related Work

HOIST is a system by Regehr [12] that can derive static analyzers for embedded systems, in their case for an Atmel ATmega16 microcontroller. This is similar to our approach. The key difference is that they do not use a description of the hardware, but either a simulator or the actual device. For a given instruction that is executed on the microcontroller, HOIST conducts an exhaustive search over all the possible inputs, and protocols the effects on the hardware. These deduced transfer functions are then compacted into binary decision diagrams (BDDs), and eventually translated into C code. While this mostly automatic approach can provide very high accuracy in instruction effects, it certainly has the disadvantage of exponential complexity in the number of parameters for an instruction. Our approach does not depend on this, and is also automated, but the correctness of the results depends on the correctness of the description of the hardware. Moreover, HOIST is limited to analyzing ALU operations, whereas our analyzer, SGDL-STA, can analyze any kind of instruction.

Chen et al. [3] have created a retargetable static analyzer for embedded software within the scope of the MESCAL project [8]. Similar to our approach, they process a description of the architecture, which in their case is called a MESCAL Architecture Description (MAD). Automatic classification of instructions for constructing the CFG is apparently also possible in their approach, and they hint at that this is possible due to some attributes present in the MAD that allow identification of, for instance, the program counter. However, no further detail is provided on the ideas involved in classification. The generated analyzer is suitable for analyzing worst case execution time of certain classes of programs intended to run on the hardware.

Schlickling and Pister [15] also analyze hardware descriptions, in their case VHDL code. Their system first translates the VHDL input into a sequential program, before it applies well-known data flow analyses such as constant propagation analysis. These analyses are then used to prove or disprove worst case execution time properties of the hardware. In contrast to this, we concentrate on the way the resource model is altered by instructions, deliberately neglecting timing.

Might [9] focuses on the step from concrete to abstract semantics for a variant of lambda calculus. In their examples, they also relate their work to register machines, which, albeit a concept from theory, share some commonalities with real-world microcontrollers. They point out the similarities between the two semantics, and how to provide analysis designers with an almost algorithmic approach to lifting the concrete to the abstract. Hence, the foremost difference to our approach is that their contribution is certainly more flexible, as they rely on an expert. Compared to this, our approach is intentionally restricted to only a few abstractions, but for these, it is fully automated.

6 Conclusion

This paper shows that a single description of an instruction-set architecture, given as an implementation in a special-purpose imperative programming language, can serve as a starting point for generating several verification tools. We have shown how to switch from register transfer-level semantics based on concrete values to a partially symbolic technique, called delayed nondeterminism. To this end, we have described static analyses used on the imperative descriptions, by which the intention behind instructions becomes visible and ready for translation. Furthermore, these analyses can also be used to obtain a characterization of instructions needed for analyzing the code for the target platform.

The concepts developed in this contribution should be applicable not only to [MC]SQUARE and the SGDL system, but to any model checker interpreting assembly code. In order to verify the concepts, we have implemented a static analyzer for SGDL, called SGDL-STA. So far, we have successfully verified the ideas concerning classification of instructions into control flow classes. Classifying the instruction sets of both the ATmega16 and the MCS-51 microcontrollers can be achieved in less than 10 seconds. Additionally, we have used the analysis results for generating an operative static analyzer for the ATmega16 simulator, which enables a variant of Dead Variable Reduction [17] for this simulator. Hence, a direction for future work will be the implementation of the other concepts, especially the creation of the abstract semantics for delayed nondeterminism, and a comparison between the derived and the manually implemented versions of this abstraction technique.

Clearly, the results indicate that abstraction for hardware-dependent model checkers can, to a certain degree, be achieved automatically. Thus, it is not strictly necessary to have an expert in both model checking and embedded systems available, who is then to perform a fine-tuning of such tools. A practical implication of this improvement is that it might be possible for a non-expert to retarget a model checker to a new platform, at least in case the set of automatically derivable abstractions suffices. Therefore, we consider it necessary to conduct further research on other abstractions, and figure out to what extent it is possible to derive their semantics as well. Obvious directions for this include lifting the concrete semantics to interval semantics (i.e., the value of a memory cell is only known to be in an interval, instead of several distinct values), and easing our restrictions on nondeterministic control flow.

7 Acknowledgments

We thank our former team member Christian Dehnert for his participation in our research, some of which was integrated into this contribution. Additionally, we thank the DFG Research Training Group 1298 "Algorithmic Synthesis of Reactive and Discrete-Continuous Systems" for funding parts of this work.

References

1. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2006.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
3. K. Chen, S. Malik, and D. August. Retargetable static timing analysis for embedded software. In *The 14th International Symposium on System Synthesis, 2001. Proceedings.*, pages 39 – 44, 2001.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL 77), Los Angeles, USA*, pages 238–252. ACM, 1977.
6. D. Gückel, J. Brauer, and S. Kowalewski. A system for synthesizing abstraction-enabled simulators for binary code verification. In *Industrial Embedded Systems (SIES 2010), Trento, Italy.*, 2010.
7. D. Gückel, B. Schlich, J. Brauer, and S. Kowalewski. Synthesizing simulators for model checking microcontroller binary code. In *13th IEEE International Symposium on Design & Diagnostics of Electronic Circuits and Systems (DDECS 2010), Vienna, Austria.*, 2010.
8. K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19:1523 –1543, 2000.
9. M. Might. Abstract interpreters for free. In R. Cousot and M. Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 407–421. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-15769-1_25.
10. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
11. T. Noll and B. Schlich. Delayed nondeterminism in model checking embedded systems assembly code. In *Hardware and Software: Verification and Testing (HVC 2007)*, volume 4899 of *LNCS*, pages 185–201. Springer, 2008.
12. J. Regehr and A. Reid. HOIST: A system for automatically deriving static analyzers for embedded systems. *ACM SIGOPS Operating Systems Review*, 38(5):133–143, 2004.
13. B. Schlich. *Model Checking of Software for Microcontrollers.* Dissertation, RWTH Aachen University, Aachen, Germany, June 2008.
14. B. Schlich, J. Brauer, and S. Kowalewski. Application of static analyses for state space reduction to microcontroller binary code. *Sci. Comput. Program.*, 76(2):100–118, 2011.
15. M. Schlickling and M. Pister. A framework for static analysis of VHDL code. In *Proceedings of 7th International Workshop on Worst-case Execution Time (WCET) Analysis*, 2007.
16. B. Titzer, J. Lee, and J. Palsberg. A declarative approach to generating machine code tools. Technical report, UCLA Computer Science Department, University of California, Los Angeles, USA, 2006.
17. K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.

Structuring Interactive Correctness Proofs by Formalizing Coding Idioms

Holger Gast

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
gast@informatik.uni-tuebingen.de

Abstract. This paper examines a novel strategy for developing correctness proofs in interactive software verification for C programs. Rather than proceeding backwards from the generated verification conditions, we start by developing a library of the employed data structures and related coding idioms. The application of that library then leads to correctness proofs that reflect informal arguments about the idioms. We apply this strategy to the low-level memory allocator of the L4 microkernel, a case study discussed in the literature.

1 Introduction

Interactive theorem proving offers several recognized benefits for functional software verification. From a foundational perspective, it enables the definition of the language semantics and the derivation of the Hoare logic, which ensures that the verification system is sound (relative to the defined semantics) (e.g. [1–3]). The background theories for reasoning about the machine model can likewise be derived, rather than axiomatized [4, §1.4][1], thus avoiding known practical issues of inconsistencies [5, §7]. From the perspective of applications, interactive provers offer strong support for the development of theories of the application domain [4, §1.3], which are not restricted to special classes of properties [6, §2.3]. In particular, they can address algorithmic considerations [7, §7.2], such as geometric questions [8, §4.4] or properties of defined predicates [9, §4.3].

However, interactive software verification incurs the obvious liability of requiring the user to guide the proof in some detail and to conceive a proof structure matching the intended correctness argument. This is the case even more in the development of background theories applicable to several algorithms. The necessity of strategic planning and human insight involved is often perceived as a major obstacle to the practical applicability of interactive proving.

This paper proposes to address the challenge of structuring correctness proofs by focusing on the idioms and coding patterns connected with the data structures found in the verified code. The benefit to be gained from this approach is clear: users can bring to bear their insight and experience as software engineers on the formal development, and the proof's structure will follow the informal correctness arguments used by developers, thus making it more understandable and hence maintainable.

We demonstrate and evaluate this strategy using a study of the memory allocator of the L4 microkernel, which has previously been verified by Tuch et al. [10, 11] and thus affords a point of comparison. Although the allocator merely maintains a sorted singly-linked list of free blocks, we have found even a simplified fragment of the code surprisingly hard to verify in a previous attempt [12], owing to the many technical aspects introduced by the low-level memory model. This impression is confirmed by the level of detail present in the original proof [13].

The benefit of the strategy proposed now can therefore be gauged by whether the found proof matches the essentially simple structure of the algorithm, thus appearing as a detailed version of an informal correctness argument. This goal also relates to a peculiarity of interactive software verification. Differing from mechanized mathematics, no effort is spent on making the proof more concise or elegant once it is found—its mechanically verified existence is sufficient.

For this reason, the paper's structure reflects the development of the proof. Section 2 gives an overview of the allocator and points out the coding idioms that make the code seem straightforward. Section 3 then formalizes these idioms in a library of singly-linked lists and their standard manipulations. Section 4 gives the correctness proof on the basis of that library, with an emphasis on the direct match between the library theorems and informal correctness arguments.

1.1 An Overview of Lightweight Separation

We conduct the proof within the lightweight separation verification system [14, 15]. It is developed as a conservative extension of Isabelle/HOL and permits the verification of low-level programs in a C dialect inspired by [1].

The idea of lightweight separation is to complement the standard formulation of assertions in HOL with explicit formal representations of the memory layout. Towards that end, assertions usually contain a conjunct $\mathsf{M} \blacktriangleright \mathsf{A}$ where M is the current memory state and A is a *cover*, which is a predicate on address sets. A cover is *well-formed* if it accepts at most one address set. We call the address set accepted by a well-formed cover A the *memory region covered by* A. For instance, the following constant captures a block of n bytes at a. It describes the address set and excludes overflows in address arithmetic, making the block contiguous ($\{\mathsf{a}..{<}\mathsf{b}\}$ denotes a half-open interval $[\mathsf{a}, \mathsf{b})$ in Isabelle/HOL; $\oplus$ is address offset).

$$\mathsf{block\,a\,n} \equiv \lambda \mathsf{S}.\ \mathsf{S} = \{\mathsf{a}\,..{<}\ \mathsf{a} \oplus \mathsf{n}\} \wedge \mathsf{a} \leq \mathsf{a} \oplus \mathsf{n}$$

$\mathsf{M} \blacktriangleright \mathsf{A}$ states A covers the allocated region of M. For $\mathsf{M} \rhd \mathsf{A}$, A is allocated in M. The *subcover* relation $\mathsf{A} \preceq \mathsf{B}$ states that the region of A is contained in the region of B. The memory layout is described by nested cover expressions combined by the *disjointness* operator $\mathsf{A} \parallel \mathsf{B}$. The system provides covers for standard constructs, such as variables and blocks whose size is given by a type. New constants for covers can be defined as needed. In particular, one can define covers for inductive data structures using Isabelle's built-in `inductive` command.

The lightweight separation tactics then prove, by symbolic manipulation of cover expressions [14], the allocatedness of memory accessed by the program and the disjointness of regions read in assertions and modified by programs. If necessary, they unfold given layouts to expose their constituent parts [15].

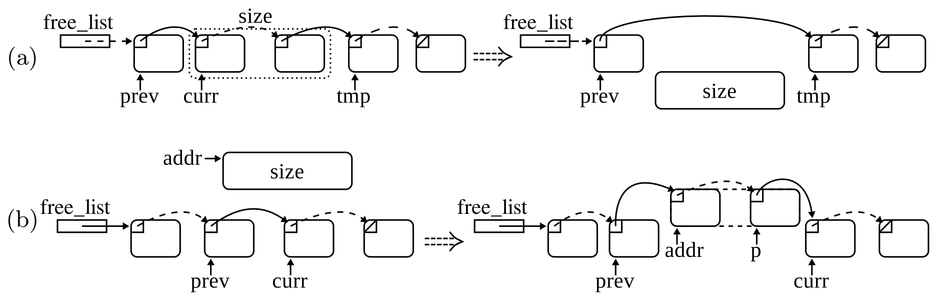

Fig. 1. Allocation and Deallocation in the Free List

2 The L4 Memory Allocator

The memory allocator of the L4 microkernel [13] is responsible for the low-level allocation of memory blocks. The interface consists of two routines `alloc` and `free` that enable client code to obtain and release memory blocks. We now describe and analyze their overall structure.

2.1 Data Structure and Routines

The microkernel allocator offers basic services to higher-level allocators and handles memory blocks as multiples of 1kb. Internally, it maintains a *free list* of 1kb *chunks*, whose first word is a pointer to the next chunk. The chunks in the list are ordered by their start addresses to enable efficient compaction during allocation. The routines `alloc` and `free` (Appendix A) in essence cut out or splice in sequences of chunks at the correct position within the free list.

The `alloc` routine advances a pointer `curr` forward through the free list (Fig. 1 (a); dashed arrows indicate the possible crossing of several chunks). At each chunk, a nested loop advances a pointer `tmp` to check whether the sequence of adjacent chunks starting at `curr` matches the requested size. If this is the case, the routine removes the sequence from the list, initializes it with 0-bytes, and returns it to the caller as a raw block of memory. The `prev` pointer is required to splice out the returned chunks and always lags one step behind the `curr` pointer.

The `free` routine dually splices a returned block of memory back into the free list (Fig. 1 (b)). Since the block's size may be a multiple of 1kb, the routine first creates a list structure inside the raw memory block. Then, it searches for the place where the new list fragment must be inserted to maintain sortedness. Finally, it links the new fragment into the free list. Like the `alloc` routine, it maintains a `prev` pointer to perform that final pointer manipulation.

2.2 Idioms for List Operations

The routines' code (Appendix A) appears straightforward after this explanation of its purpose. The reason for the simple reading is that the code only applies

```
ListNode *p = list_head;
while (p != NULL && not found) {
  perform check & return/break;
  p = p->next;
}
```

(a)

```
ListNode **prev = &list_head;
ListNode *p = list_head;
while (p != NULL && not found) {
  perform check & return/break;
  prev = &p->next;
  p = p->next;
}
```

(b)

Fig. 2. Iteration Through Lists

well-known patterns and idioms: the experienced developer recognizes these and uses them in arguments about the code's functionality and correctness.

The first and most basic idiom is the search for a particular point in a list. The coding pattern is shown in Fig 2 (a): some pointer `p` is initialized to the first node, and is advanced repeatedly by dereferencing the `next` pointer of the node. The checks in the while test or body are usually used alternatively. Following the terminology of the C++ STL [16], we will call the pointer `p` an *iterator*. Informally, the iteration works without failure because `p` never leaves the list structure: it is "properly" initialized and "properly" advanced to the next node in the list. Since it points to a node after the test for `NULL`, the iterator `p` can be dereferenced in the checks without causing a memory fault.[1]

If a modification is intended after the search, the idiom must be extended by some reference to the predecessor node of `p`, in order to insert or remove nodes at `p` by pointer manipulation. There are several variants of such an extension. The L4 allocator uses the one shown in Fig. 2 (b), which makes use of C's ability of taking the addresses of arbitrary memory objects. The constraints associated with `prev` are that `*prev` $=$ `p` and that `prev` points either to the list-head variable or to the `next` field of some node in the list.[2]

After the loop, the manipulation of the list structure involves the assignment `*prev =` q, where q is either some successor node of `p` for the removal of `p` or a new node to be inserted before `p`. To show that the resulting pointer structure is the desired linked list, informal arguments usually use pointer diagrams: the reached situation is shown in Fig. 3 on the left. If `prev` $=$ `&head`, the argument is simple. Otherwise, one needs to expose the node containing the `prev` pointer in the diagram, possibly followed by extracting node `p` from the remaining list. Then, one draws the algorithm-specific pointer operations, e.g. those of Fig. 1, and argues that the expected list structure results. Note that the case distinction on `prev` $=$ `&head`, which is necessary in the argument, is not present in the code.

[1] These are also the requirements for the STL's most basic *forward iterator* [16].

[2] A common alternative uses a sentinel head node, such that `prev` is a node pointer and `p` is inlined as `prev->next` (e.g. the `slist` library of `g++`). This variant has the advantage of unifying the reasoning by avoiding the case distinction about the exact target of `prev`. The library in §3 supports this variant as well.

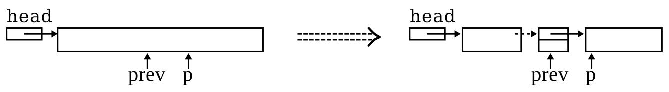

Fig. 3. Extraction of a Node in the Follow Iterator Idiom

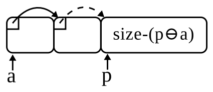

Fig. 4. Establishing the Successor Structure

2.3 Idioms for Aligned Low-level Memory Access

The iterator idiom, once identified, can also be applied to algorithms that do not handle list-like data structures. The `zero_mem` function (Appendix A), for example, initializes a memory block by writing machine words, i.e. by setting groups of 4 bytes at a time. Its loop advances the pointer `(int*)p+i` as an iterator in steps of 4, by incrementing `i` in each loop iteration. The pointer is properly initialized by setting `i=0`. Advancing the iterator by `i++` leaves it within the bounds of the raw memory block, because the block's size is a multiple of 4.

The first loop of `dealloc` (§4.3) similarly establishes a list structure in a memory block (Fig. 4) by advancing a pointer `p`, initialized to start address `a`, in 1kb-steps. The proof obligations are the same as for the iterator in `zero_mem`.

The correctness arguments in both cases therefore consist of the familiar "initializing" and "advancing" an iterator, and can be carried out analogously to the list case. Although the formulation of the invariants is quite different, the proofs thus still reflect the common structure. For space reasons, we will not discuss them further.

3 A Library of List Manipulations

This section captures the idioms from §2.2 in a generic library of singly-linked lists. Using this library, the correctness proof in §4 will be structured according to informal arguments, after the allocator's free list has been proven an instance of the general case. For space reasons, we omit derived constructs, such as the typed abstraction of the nodes' contents as HOL values and the variant of follow iterators mentioned in Footnote 2. The library consists of 750 lines and has been re-used in two further case studies (§6).

3.1 Parameters and Assumptions

The library is formulated as an Isabelle locale [17] to abstract over the specific structure of list nodes. Locales can depend on parameters and state assumptions about these. The list library has three parameters (1) ("::" denotes a type constraint): `node` is a cover (§1.1) for a single list node and `succ` reads the successor (or "next"-) link from a given node in a given memory state. Both usually depend on type definitions, so a global context gctx with these definitions is added.

$$\mathsf{node} :: \mathsf{addr} \Rightarrow \mathsf{cover} \qquad \mathsf{succ} :: \mathsf{addr} \Rightarrow \mathsf{memory} \Rightarrow \mathsf{addr} \qquad \mathsf{gctx} :: \mathsf{ctx} \tag{1}$$

$$\mathsf{accesses}\,(\mathsf{succ}\,\mathsf{p})\,\mathsf{M}\,(\mathsf{node}\,\mathsf{p}) \qquad \mathsf{node}\,\mathsf{p}\,\mathsf{S} \Longrightarrow \mathsf{p} \in \mathsf{S} \qquad \mathsf{wf\text{-}cover}\,(\mathsf{node}\,\mathsf{p}) \tag{2}$$

The theory makes three natural assumptions (2) about these parameters: reading the successor of a node depends only on that node ($\mathsf{accesses}\,\mathsf{f}\,\mathsf{M}\,\mathsf{A}$ states that memory-reading function f, when applied to state M, depends only on the region covered by A); the base pointer of a node is contained in its footprint; finally, the node cover must be well-formed (§1.1). Note that these assumptions are implicit in pointer diagrams and are validated by the usual list structures in programs.

3.2 List Structure

The standard approach to lists (e.g. [18]) is to define a predicate to enumerate the nodes in list fragments. An inductive definition is given by the introduction rules (3). A parallel definition for $\mathsf{cover}\,\mathsf{p}\,\mathsf{q}\,\mathsf{M}$, the memory region of the list, is straightforward. ($[]$ is the empty list, # denotes the "cons" operation)

$$\frac{\mathsf{p} = \mathsf{q} \quad \mathsf{xs} = [\,]}{\mathsf{nodes}\ \mathsf{p}\ \mathsf{q}\ \mathsf{xs}\ \mathsf{M}} \qquad \frac{\mathsf{p} \neq \mathsf{q} \quad \mathsf{nodes}\ (\mathsf{succ}\ \mathsf{p}\ \mathsf{M})\ \mathsf{q}\ \mathsf{ys}\ \mathsf{M} \quad \mathsf{xs} = \mathsf{p}\ \#\ \mathsf{ys}}{\mathsf{nodes}\ \mathsf{p}\ \mathsf{q}\ \mathsf{xs}\ \mathsf{M}} \tag{3}$$

Already at this point, the library yields a benefit in the form of useful properties, such as the nodes of a list being distinct (4).

$$\frac{\mathsf{nodes}\ \mathsf{p}\ \mathsf{q}\ \mathsf{xs}\ \mathsf{M}}{\mathsf{distinct}\ \mathsf{xs}} \tag{4}$$

Due to their parallel definitions, the nodes and the cover of a list are closely related. In particular, if a list is allocated, then it consists of a sequence of nodes (5) and—since `null` is never allocated—it cannot contain `null` as a node (6).

$$\frac{\mathsf{M} \rhd \mathsf{cover}\ \mathsf{p}\ \mathsf{q}\ \mathsf{M}}{\exists \mathsf{xs}.\ \mathsf{nodes}\ \mathsf{p}\ \mathsf{q}\ \mathsf{xs}\ \mathsf{M}} \tag{5}$$

$$\frac{\mathsf{M} \rhd \mathsf{node}\ \mathsf{p}}{\mathsf{p} \neq \mathsf{null}} \qquad \frac{\mathsf{nodes}\ \mathsf{p}\ \mathsf{q}\ \mathsf{xs}\ \mathsf{M} \quad \mathsf{M} \rhd \mathsf{cover}\ \mathsf{p}\ \mathsf{q}\ \mathsf{M}}{\mathsf{null} \notin \mathsf{set}\ \mathsf{xs}} \tag{6}$$

We have noted in §2 that informal arguments by pointer diagrams address the "extraction" of nodes from a list and the resulting "overall" list. We now reflect the graphical arguments in the form of theorems to make their application straightforward: for every change in the pointer diagram, the formal proof contains an application of the corresponding theorem. For space reasons, we omit the parallel development for cover.

Theorems (7) and (8) enable the extraction and integration of the first node of a list. Note how the pre-condition $\mathsf{p} \neq \mathsf{q}$ reflects the check of the idiomatic while loops from §2.2. To save a separate application of (4), (7) yields the derived information that the nodes were originally distinct. The complementary theorems (9) and (10) manipulate the last node of a list. The final rules (11) and (12) reflect splitting and joining at a given node of the list, as is necessary for Fig. 1. The last premises of (10) and (12) ensure that no cycles have been created. In the frequent case where q is null, they can be proven by (6); the library provides specialized rules for this case to simplify proofs further.

$$\frac{\mathsf{p} \neq \mathsf{q}}{\mathsf{nodes\ p\ q\ xs\ M} = (\exists\, \mathsf{ys.\ nodes\ (succ\ p\ M)\ q\ ys\ M} \wedge \mathsf{xs} = \mathsf{p} \# \mathsf{ys} \wedge \mathsf{p} \notin \mathsf{set\ ys})} \tag{7}$$

$$\frac{\mathsf{nodes\ r\ q\ ys\ M} \quad \mathsf{succ\ p\ M} = \mathsf{r} \quad \mathsf{p} \neq \mathsf{q}}{\mathsf{nodes\ p\ q\ (p} \# \mathsf{ys)\ M}} \tag{8}$$

$$\frac{\mathsf{q} = \mathsf{succ\ r\ M} \quad \mathsf{r} \in \mathsf{set\ xs}}{\mathsf{nodes\ p\ q\ xs\ M} = (\exists\, \mathsf{ys.\ nodes\ p\ r\ ys\ M} \wedge \mathsf{xs} = \mathsf{ys} @ [\,\mathsf{r}\,] \wedge \mathsf{q} \notin \mathsf{set\ xs})} \tag{9}$$

$$\frac{\mathsf{nodes\ p\ r\ ys\ M} \quad \mathsf{succ\ r\ M} = \mathsf{q} \quad \mathsf{q} \notin \mathsf{set\ (ys} @ [\mathsf{r}])}{\mathsf{nodes\ p\ q\ (ys} @ [\mathsf{r}])\ \mathsf{M}} \tag{10}$$

$$\frac{\mathsf{r} \in \mathsf{set\ xs}}{\mathsf{nodes\ p\ q\ xs\ M} = (\exists\, \mathsf{ys\ zs.\ nodes\ p\ r\ ys\ M} \wedge \mathsf{nodes\ r\ q\ zs\ M} \wedge \mathsf{xs} = \mathsf{ys} @ \mathsf{zs} \wedge \mathsf{q} \notin \mathsf{set\ xs})} \tag{11}$$

$$\frac{\mathsf{nodes\ p\ q\ xs\ M} \quad \mathsf{nodes\ q\ r\ ys\ M} \quad \mathsf{r} \notin \mathsf{set\ xs}}{\mathsf{nodes\ p\ r\ (xs} @ \mathsf{ys)\ M}} \tag{12}$$

3.3 Iterators

In principle, the definitions and theorems from §3.2 are sufficient state loop invariants and perform proofs about list manipulating programs (e.g. [18]). However, this approach invariably has to consider the set of list nodes. The idioms of §2.2, on the other hand, focus on the "current" and the "next" node, which aligns the informal reasoning with the local list structure—the inductive argument about the iterator referencing one of the list's nodes is left implicit.

We can obtain proofs that reflect the informal reasoning by formalizing the idea of an "iterator" itself. In the STL concept [16], an iterator into some data structure always points to one of its elements or is a special one-past-the-end iterator. In the case of fragments of singly-linked lists, this idea is expressed by the following definition.

$\mathsf{iter\ a\ p\ q\ M} \equiv \exists \mathsf{xs.\ nodes\ p\ q\ xs\ M} \wedge (\mathsf{a} \in \mathsf{set\ xs} \vee \mathsf{a} = \mathsf{q})$

The loop invariant for the iteration idiom can then simply contain the conjunct $\mathsf{iter\ p\ head\ null\ M}$, thus hiding the list structure as desired. Furthermore, the informal arguments about "initializing" and "advancing" an iterator from §2.2 are reflected by theorems (13), and these are used to establish the invariant and prove its preservation after the loop body. When the sought node in the list has been found, it can be exposed by (14), followed by (8), without leaving the iterator idiom.

$$\frac{\mathsf{M} \rhd \mathsf{cover\ p\ q\ M}}{\mathsf{iter\ p\ p\ q\ M}} \qquad \frac{\mathsf{iter\ a\ p\ q\ M} \quad \mathsf{a} \neq \mathsf{q}}{\mathsf{iter\ (succ\ a\ M)\ p\ q\ M}} \tag{13}$$

$$\frac{\mathsf{iter\ r\ p\ q\ M}}{\mathsf{nodes\ p\ q\ xs\ M} = (\exists\, \mathsf{ys\ zs.\ nodes\ p\ r\ ys\ M} \wedge \mathsf{nodes\ r\ q\ zs\ M} \wedge \mathsf{xs} = \mathsf{ys} @ \mathsf{zs} \wedge \mathsf{q} \notin \mathsf{set\ xs})} \tag{14}$$

3.4 Follow Iterators

Whenever a list manipulation is intended after iteration, one has to keep an auxiliary pointer to the node preceding the current one (§2.2). Since the pattern is so frequent, we introduce another abstraction to capture it. Since the prev pointer lags one step behind a cur pointer, we choose the term *follow iterator*.

The locale for follow iterators extends that of iterators by introducing parameters that abstract over the structure of the "successor field", i.e. the memory object containing the "next" pointer. By the idiom, this structure must be the same as that of the head variable. The structure is given by a cover succ-field. The function rd-succ-field is used for reading its content. The offset of the field within the node is given by succ-field-off.

```
succ-field     :: "addr ⇒ cover"
succ-field-off :: "word32"
rd-succ-field  :: "addr ⇒ memory ⇒ addr"
```

The locale's assumptions describe the following relationships between these parameters: the special accessor reads the information gained by succ (§3.2) and depends only on the given region, which must be contained in the corresponding list node and must be well-formed (§1.1).

$$\text{rd-succ-field}\,(\text{p} \oplus \text{succ-field-off})\ \text{M} = \text{succ p M}$$

$$\text{accesses}\,(\text{rd-succ-field p})\ \text{M}\ (\text{succ-field p})$$

$$\text{succ-field p} \preceq \text{node}\,(\text{p} \oplus \text{- succ-field-off}) \qquad \text{wf-cover}\,(\text{succ-field p})$$

The follow iterator abstraction is then defined as expected (§2.2, Fig. 1): cur is an iterator, while prev points to a link to cur; further, prev either points to the head variable or is itself an iterator within the list.

$$\begin{aligned}&\text{follow-iter prev cur head p q M} \equiv\\ &\quad \text{iter cur p q M} \wedge \text{rd-succ-field prev M} = \text{cur} \wedge\\ &\quad (\text{prev} = \text{head} \vee (\text{prev} \oplus \text{- succ-field-off} \neq \text{q} \wedge \text{iter}\,(\text{prev} \oplus \text{- succ-field-off})\ \text{p q M})\end{aligned}$$

This newly defined construct establishes another layer of abstraction over the raw list structure, in that it enables the now familiar reasoning patterns in a self-contained system: theorems (15) and (16) capture the initializing and advancing of the iterator and thus replace (13) in the proofs. It is worth checking that the additional premises reflect the initializations from the idiomatic code (§2.2, Fig. 2 (b)), thus making the application of theorems straightforward.

$$\frac{\text{M} \rhd \text{cover p q M} \quad \text{prev} = \text{head} \quad \text{cur} = \text{p} \quad \text{cur} = \text{rd-succ-field prev M}}{\text{follow-iter prev cur head p q M}} \tag{15}$$

$$\frac{\begin{array}{c}\text{follow-iter prev' cur' head p q M}\\ \text{cur'} \neq \text{q} \quad \text{cur} = \text{succ cur' M} \quad \text{prev} = \text{cur'} \oplus \text{succ-field-off}\end{array}}{\text{follow-iter prev cur head p q M}} \tag{16}$$

Furthermore, the reasoning about the modification after the search from Fig. 3 can now be expressed in a single theorem (17). The prerequisite case distinction from the informal argument of §2.2 can be introduced by the (tautological) rule (18) by a single tactic invocation, saving explicit terms in the proof.

$$\frac{\text{follow-iter prev cur head p q M} \qquad \text{prev} \neq \text{head}}{\begin{array}{ll}\text{nodes p q xs M} = & (\exists \text{ys zs. nodes p prev ys M} \wedge \text{ nodes cur q zs M} \wedge \\ & \quad \text{xs} = \text{ys @ prev \# zs} \wedge \text{q} \notin \text{set xs})\end{array}} \tag{17}$$

$$\frac{\text{follow-iter prev cur head p q M}}{\text{prev} = \text{head} \vee \text{prev} \neq \text{head}} \tag{18}$$

The follow-iter abstraction thus encapsulates all information necessary to perform the split. This is evident in the proof of (17), which is based on a combination of the elementary lemmas (11), (12), (7), and (4) about the list structure (§3.2). While that proof still follows an informal argument by pointer diagram, the formalization in follow-iter and (17) enables the user to link the concrete proof to the code's intention directly. Furthermore, it saves a lot of detailed and cumbersome manipulation of formulae, which we struggled with in [12], and makes the proof more readable and thus more maintainable.

4 The Correctness Proof

This section gives the correctness proof of the allocator. The proof is structured by the application of the library from §3 and thus follows the informal arguments used in §2. The proof script is available from the author's homepage [19].

4.1 Formalizing the Allocator's Free List

We first instantiate the list library from §3 for the allocator's free list. Even though the library seems to suggest some "typed" concept of lists, the allocator's data structure fits directly: after instantiating the parameters as follows and discharging the library's assumptions by 40 lines of straightforward tactics, the developed constants and reasoning patterns are available. («» delineates program syntax in HOL, here that of types. The system contains a pre-processor.)

```
node p            ≡ block p 1024
succ p M          ≡ to-ptr (rd gctx p «void*» M)
succ-field p      ≡ typed-block gctx p «void*»
rd-succ-field a M ≡ to-ptr (rd gctx a «void*» M)
succ-field-off    ≡ 0
```

We then introduce an abbreviation kfree-list for reading the global head variable `kfree_list` and define the invariant free-list-inv: the chunks in the list are ordered by their base addresses and they are aligned to 1kb. The free-list-cover summarizes the memory occupied by the data structure.[3]

```
kfree-list ctx M       ≡ to-ptr (rdv (in-globals ctx) "kfree-list" M)
free-list-inv ctx M    ≡ (∃C. nodes (kfree-list ctx M) null C M ∧ sorted C ∧
                              (∀p ∈ set C. aligned p 1024))
free-list-cover ctx M  ≡ var-block (in-globals ctx) "kfree-list" || cover (kfree-list ctx M) null M
```

[3] We note in passing that the introduced information hiding is maintained for clients by the theorem accesses (free-list-inv ctx) M (free-list-cover ctx M): the lightweight separation framework will prove that the free list is not influenced by the clients' memory manipulations and thus solves the frame problem (e.g. [20]) in a natural fashion.

4.2 Allocation

The alloc routine searches for a contiguous block of memory that is large enough to fit the requested size (§2.1). Its specification is translated from [13]: the pre-condition requires that the free list data structure is intact and the memory does contain the free list. Furthermore, the requested size must be a multiple of 1kb.

$$\mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \wedge \mathsf{free\text{-}list\text{-}inv\ ctx\ M} \wedge 0 < \mathsf{size} \wedge 1024\ \mathsf{udvd\ size} \wedge \mathsf{size} = \mathsf{SIZE}$$

The post-condition distinguishes between success and failure. In both cases, the data structure itself is preserved. If the allocation is successful, an aligned block of 0-initialized memory has been extracted from the free list. The auxiliary (or logical) variable SIZE links the pre- and post-conditions as usual.

$$\begin{array}{l}\mathsf{free\text{-}list\text{-}inv\ ctx\ M} \wedge \\ (\mathsf{return} \neq \mathsf{null} \longrightarrow \mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \parallel \mathsf{block\ return\ SIZE} \wedge \\ \qquad\qquad\qquad\qquad \mathsf{aligned\ return\ 1024} \wedge \mathsf{zero\text{-}block\ ctx\ return\ SIZE\ M}) \wedge \\ (\mathsf{return} = \mathsf{null} \longrightarrow \mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M})\end{array}$$

The nested loops (Appendix A) of alloc advance the pointers curr and tmp, where the inner loop leaves curr unchanged. The outer loop invariant is therefore the same as the following inner loop invariant, except that Lines 3–4 are missing:

1 $\mathsf{free\text{-}list\text{-}inv\ ctx\ M} \wedge \mathsf{size} = \mathsf{SIZE} \wedge 0 < \mathsf{size} \wedge 1024\ \mathsf{udvd\ size} \wedge \mathsf{i} \leq \mathsf{size\ div\ 1024} \wedge$
2 $\mathsf{follow\text{-}iter\ prev\ curr}$ «&kfree-list» $\mathsf{kfree\text{-}list\ null\ M} \wedge$
3 $\mathsf{curr} \neq \mathsf{null} \wedge$
4 $(\mathsf{tmp} \neq \mathsf{null} \longrightarrow \mathsf{cover\ curr\ tmp\ M} = \mathsf{block\ curr\ (i * 1024)} \wedge \mathsf{iter\ tmp\ curr\ null\ M}) \wedge$
5 $\mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \parallel \mathsf{size} \parallel \mathsf{prev} \parallel \mathsf{curr} \parallel \mathsf{tmp} \parallel \mathsf{i} \wedge$
6 $\mathsf{M} \rhd \mathsf{typed\text{-}block\ prev}$ «void*» $\parallel \mathsf{size} \parallel \mathsf{prev} \parallel \mathsf{curr} \parallel \mathsf{tmp} \parallel \mathsf{i}$

Line 1 preserves the pre-condition and states that i will not exceed the bound given by the size parameter. Line 2 invokes the follow iterator idiom (§2.2) from the library (§3.4). Line 3 preserves the test result of the outer loop. Line 4 uses the notation for memory layouts (§1.1, §3.2) to state that a contiguous block of memory is found between curr and tmp. Line 5 extends the initial memory layout by the local variables; Line 6 adds that prev is not a local variable while leaving open whether it refers to the variable kfree-list or a list node.

The structure of the correctness proof is now already clear: the initializations before both loops leave precisely the situation where theorems (13) and (15) about the initialization of iterators apply. For the preservation of the outer invariant, the pointer assignments in the body match the idiom (§2.2) such that (16) is sufficient. All of these steps thus reflect the idiomatic, informal view, and the proof is merely a more precise form of argument.

For the preservation of the inner invariant, the then-branch is trivial. In the else-branch, only Line 4 needs to be newly established. In the conceptual view of §2.2, the code advances the iterator tmp; correspondingly (13) solves the iter-part immediately. The remainder of Line 4 contains the core of the algorithm: we have to prove that the found block is still contiguous, using that $\mathsf{tmp} = \mathsf{curr} + \mathsf{i} * 1024$ by the if-test. Fig. 5 depicts the proof obligation, using primed variables for the pre-state of the loop body. The figure also contains the idea of the proof: on the right-hand side of the equation from Line 4, we split off one chunk at the end of the list by (9); on the left-hand side, we split the contiguous block at address tmp'. This strategy can be expressed by 8 lines of tactics.

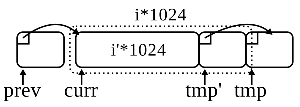

Fig. 5. Extending the Found Block with a New Chunk

The final proof obligation concerns the returning of an allocated memory block after the inner loop. Here, the assignment `*prev=tmp` splices out a sequence of chunks (Fig. 1). Since that assignment matches the idiom from §2.2, we can use (17) to perform the split of the list, after a case distinction by (18). Then, Line 4 of the invariant yields the memory layout of the post-condition. The argument takes 30 lines of tactics for both cases together; the application of the theorems reflects the informal manipulation of pointer diagrams in all steps.

4.3 Deallocation

The `free` routine takes a block of memory and integrates it into the allocator's free list (Fig. 1). Its specification, again translated from [13], requires that the free list is intact and allocated and that the block's size is a multiple of 1kb.

$\mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \parallel \mathsf{block\ a\ size} \wedge \mathsf{free\text{-}list\text{-}inv\ ctx\ M} \wedge$
$0 < \mathsf{size} \wedge 1024\ \mathsf{udvd\ size} \wedge \mathsf{aligned\ a}\ 1024$

It guarantees that the passed block has been merged into the free list.

$\mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \wedge \mathsf{free\text{-}list\text{-}inv\ ctx\ M}$

The function `free` consists of two loops. The first one establishes the pointer structure within the passed memory block, the second splices the created list into the free list at the correct position.

The first loop uses an iterator-like construct to establish the list structure within the raw memory block (§2.3; Fig. 4). We have developed a thin wrapper around Isabelle's Word library [21] to enable the idiomatic reasoning about initializing and advancing iterators. The proof that the overall block maintains the shape of Fig. 4, i.e. an initial list of elements with a trailing raw block, can be proven along the graphical intuition, by using essentially the same steps as the derivation from Fig. 5 in §4.2.

The invariant of the second loop is again typical of a search loop (§2.2, §3.4):

1 $\exists \mathsf{B}.\ \mathsf{M} \blacktriangleright \mathsf{free\text{-}list\text{-}cover\ ctx\ M} \parallel \mathsf{cover\ a\ p\ M} \parallel \mathsf{node\ p} \parallel \mathsf{a} \parallel \mathsf{size} \parallel \mathsf{p} \parallel \mathsf{prev} \parallel \mathsf{curr} \wedge$
2 $\quad \mathsf{free\text{-}list\text{-}inv\ ctx\ M} \wedge 0 < \mathsf{size} \wedge 1024\ \mathsf{udvd\ size} \wedge \mathsf{aligned\ a}\ 1024 \wedge \mathsf{aligned\ p}\ 1024 \wedge$
3 $\quad \mathsf{cover\ a\ p\ M} \parallel \mathsf{node\ p} = \mathsf{block\ a}\ (\mathsf{p} \oplus 1024 \ominus \mathsf{a}) \wedge$
4 $\quad \mathsf{nodes\ a\ p\ B\ M} \wedge \mathsf{sorted\ B} \wedge (\forall\ \mathsf{b} \in \mathsf{set\ B}.\ \mathsf{a} \leq \mathsf{b} \wedge \mathsf{b} < \mathsf{p} \wedge \mathsf{aligned\ b}\ 1024) \wedge$
5 $\quad \mathsf{follow\text{-}iter\ prev\ curr}\ «\&\mathsf{kfree\text{-}list}»\ \mathsf{kfree\text{-}list\ nill\ M} \wedge$
6 $\quad (\mathsf{prev} = «\&\mathsf{kfree\text{-}list}» \vee \mathsf{prev} < \mathsf{a}) \wedge \mathsf{a} \leq \mathsf{p}$

Lines 1–2 maintain the information of the pre-condition; Lines 3–4 keep the result of the first loop (Fig. 4). Line 5 captures `curr` as a follow iterator (§3.4) for the search, while Line 6 characterizes the nodes that `curr` has already passed as having strictly smaller start addresses that `a`.

Since the loop matches the idiom (Fig. 2 (b)), its correctness proof follows the reasoning already discussed for `alloc` in §4.2: (15) and (16) yield initialization and preservation; Line 6 follows from the while-test. After the loop, the new sequence of nodes `a`...`p` is spliced into the free list before `curr`, again making use of (17) and (18) to split the overall list structure before the pointer updates.

5 Related Work

To the best of the author's knowledge, the proposal of developing background theories by formalizing idioms and coding patterns has not been discussed previously. We therefore focus on similar case studies and on approaches to structuring interactive proofs beyond the discharging of generated verification conditions.

Tuch et al. [10, 13] give two proofs of the L4 memory allocator, one using separation logic and one using a typed view on the raw memory. Their development shows the intricacy of reasoning about byte-addressed finite memory. Our own proof clearly benefits from Isabelle's Word library [21] contributed by the L4 verification project. In his analysis [11, §6.6], Tuch suggests that with further experience in similar proofs, a set of re-usable libraries could be constructed to aid in future developments. He proposes to collect lemmas that have been found useful, and to improve automation for separation logic assertions. Differing from ours, his approach is thus goal-directed, starting from the verification conditions. Although proof sizes in different systems are not directly comparable, it is interesting that our proof is significantly shorter (by a factor of 2) even though Tuch et al. prove only the immediately necessary theorems.

Marti et al. [22] verify the heap manager of the Topsy operating system, which is also based on an untyped singly-linked list. The paper focuses on the developed verification environment and therefore the actual proof is discussed only at the level of the defined predicates and the function specifications. An instance of forward reasoning appears in [22, §4.2], where a central theorem for compacting two list nodes is derived beforehand and is shown to apply to an example Hoare triple of an expected format. The structure of the greater part of the proof ($\approx$4500 lines of Coq) is not analyzed further.

Böhme et al. [4] investigate the advantages of interactive theorem proving for software verification. In [4, §1.3], they observe that the introduction of suitable abstractions with well-developed theories can make interactive proofs feasible where automated provers fail because they have to unfold the definitions. They demonstrate the claim by a case study on an implementation of circular singly-linked lists, but do not formulate strategies to develop general theories.

Concerning the question of structuring interactive correctness proofs, Myreen [23, §5.2] verifies Cheney's garbage collector using a refinement argument. The first two layers capture the specification and abstract implementation of copying garbage collection; they can thus be read as the common structure of different collectors. Our proposal of formalizing idioms addresses, on the other hand, cross-cutting issues of different algorithms. McCreight's proof [24] of the same algorithm introduces carefully chosen separation logic predicates that reflect the structure of pointer diagrams, and diagrammatic arguments are used to illustrate the proof strategies. However, their translation into a proof script involves a substantial amount of technical formula manipulation [24, §6.3.3, p. 122, §6.4.3]. Both the defined predicates and the proof strategies are specific to the algorithm.

A different approach to interactive proving has been proposed by Tuerk [25] and Chlipala [3]. They use a restricted form of separation logic, inspired by Smallfoot [26]. Besides pure assertions, verification conditions then consist of implications between iterated spatial conjunctions, which are cancelled syntactically one-by-one, possibly using user-supplied unfolding rules. This process reduces the verification conditions to pure assertions, which are solved mostly automatically by the built-in tactics of the interactive provers.

6 Conclusion

Interactive software verification enables the development of theories independently of concrete verification conditions, with a view to making proofs readable, maintainable, and possibly re-usable. This paper has proposed to structure such theories around the idioms and coding patterns employed by developers, and to formulate the definitions and theorems to reflect informal arguments about the code, e.g. in the form of pointer diagrams.

We have demonstrated this strategy using the frequent case of singly-linked lists. Besides their basic structure (§3.2), we have introduced the higher-level idioms of *iterators* (§3.3) for read-only searches and *follow iterators* (§3.4) for searching and modifying lists. The developed library is formulated as an Isabelle locale and can be instantiated for different concrete list structures. We have applied the library to the untyped free list of the L4 memory allocator [10, 13]. It was interesting to find during the development that the reasoning patterns embodied in the library made the overall proof [19] much more straightforward than the previous partial attempt [12], even though several additional points, such as alignment and the initialization of allocated memory had to be considered.

The proposed strategy has shown several benefits: first, all verification conditions regarding the list structure were solved by library theorems, and their application in each case reflected informal arguments by pointer diagrams. The chosen theorem names preserve this link in the proof script [19], thus contributing to its maintainability. Second, the analogies between the allocator's routines could be exploited by having a common ground for expressing them (§4.2, §4.3). Third, although no specific effort was made, the script is substantially smaller than the original one [13, 11], which can be attributed to the simple application of library theorems due to their matching the coding idioms.

Finally, the library's genericity has enabled its re-use for the work queues of the Schorr-Waite graph marking algorithm [15] and Cheney's collector [27, 28]. Both algorithms use a non-standard successor link, involving a case-distinction and pointer arithmetic, respectively. The correctness proofs are nevertheless covered by the library theorems (§3.2). Between the two algorithms, we have re-used a theory of object graphs [15, §5.1] that is also structured around expected common manipulations. This further example suggests that the strategies proposed now will be applicable beyond the chosen case study.

A Source Code

```
void *alloc(unsigned int size) {
  void **prev = &kfree_list;
  void *curr = kfree_list;
  while (curr != null) {
    void *tmp = *(void **)curr;
    unsigned int i = 1;
    while (tmp != null &&
           i < size / 1024) {
      if (tmp != curr + i * 1024) {
         tmp = null;
      } else {
         tmp = *(void**)tmp;
         i++;
      }
    }
    if (tmp != null) {
      *prev = tmp;
      zero_mem(curr,size);
      return curr;
    }
    prev = (void**)curr;
    curr = *(void**)curr;
  }
  return null;
}
```

```
void free(void *a, unsigned int size) {
  void *p;
  void **prev;
  void *curr;
  p = a;
  while (p < a + (size - 1024)) {
    *(void**)p = p + 1024;
    p = *(void**)p;
  }
  prev = &kfree_list;
  curr = kfree_list;
  while (curr != null && (a > curr)) {
    prev = (void**)curr;
    curr = *(void**)curr;
  }
  *prev = a;
  *(void**)p = curr;
}

void zero_mem(void *p, unsigned int n) {
 unsigned int i = (unsigned int)0;
 while (i < n / 4) {
   *((int*)p+i) = 0;
   i++;
 }
}
```

References

1. Norrish, M.: C formalised in HOL. PhD thesis, University of Cambridge (1998) Technical Report UCAM-CL-TR-453.
2. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2005)
3. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI'11). (2011)
4. Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie–An interactive prover-backend for the Verifying C Compiler. J. Autom. Reason. **44** (2010) 111–144
5. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: a verification experience report. In Shankar, N., Woodcock, J., eds.: VSTTE'08. Volume 5295 of LNCS., Springer (2008) 177–191
6. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM (CACM) **53**(6) (2010) 107–115
7. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. SIGPLAN Not. **43**(6) (2008) 349–361
8. Walter, C.L.D.: Certifiable specification and verification of C programs. In: Formal Methods. Formal Methods (FM 2009), Springer (2009)
9. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In Aichernig, B.K., Beckert, B., eds.: SEFM, IEEE (2005)

10. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In Hofmann, M., Felleisen, M., eds.: Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07). (2007)
11. Tuch, H.: Formal Memory Models for Verifying C Systems Code. PhD thesis, School of Computer Science and Engineering, University of NSW (2008)
12. Gast, H., Trieflinger, J.: High-level Reasoning about Low-level Programs. In Roggenbach, M., ed.: Automated Verification of Critical Systems 2009. Volume 23 of Electronic Communications of the EASST., EASST (2009)
13. Tuch, H., Klein, G., Norrish, M.: Verification of the L4 kernel memory allocator. formal proof document. Technical report, NICTA (2007) `http://www.ertos.nicta.com.au/research/l4.verified/kmalloc.pml`.
14. Gast, H.: Lightweight separation. In Ait Mohamed, O., Munoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics 21st International Conference (TPHOLs 2008). Volume 5170 of LNCS., Springer (2008)
15. Gast, H.: Reasoning about memory layouts. Formal Methods in System Design **37**(2-3) (2010) 141–170
16. Austern, M.H.: Generic Programming and the STL — using and extending the C^{++} Standard Template Library. Addison-Wesley (1998)
17. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales—A sectioning concept for Isabelle. In Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L., eds.: TPHOLs. Volume 1690 of LNCS., Springer (1999) 149–166
18. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. **199**(1–2) (2005) 200–227
19. Gast, H.: Verifying the L4 kernel allocator in lightweight separation (2010) `http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/kalloc.pdf`.
20. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of LNCS., Springer (2006) 268–283
21. Dawson, J.E.: Isabelle theories for machine words. In: 7th International Workshop on Automated Verification of Critical Systems (AVOCS'07). Volume 250 of ENTCS. (2009)
22. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In Liu, Z., He, J., eds.: ICFEM. Volume 4260 of LNCS., Springer (2006) 400–419
23. Myreen, M.O.: Formal verification of machine-code programs. PhD thesis, University of Cambridge (2009) UCAM-CL-TR-765.
24. McCreight, A.: The Mechanized Verification of Garbage Collector Implementations. PhD thesis, Department of Computer Science, Yale University (2008)
25. Tuerk, T.: A formalisation of Smallfoot in HOL. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009). Volume 5674 of LNCS., Springer (2009)
26. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS., Springer (2005)
27. Gast, H.: Developer-oriented correctness proofs: A case study of Cheney's algorithm. In Qin, S., Qiu, Z., eds.: Proceedings of 13th International Conference on Formal Engineering Methods (ICFEM 2011). LNCS, Springer (2011) (to appear).
28. Gast, H.: A developer-oriented proof of Cheney's algorithm (2011) `http://www-pu.informatik.uni-tuebingen.de/users/gast/proofs/cheney.pdf`.

Verification of Dependable Software using SPARK and Isabelle

Stefan Berghofer[*]

secunet Security Networks AG
Ammonstraße 74, 01067 Dresden, Germany

Abstract. We present a link between the interactive proof assistant Isabelle/HOL and the SPARK/Ada tool suite for the verification of high-integrity software. Using this link, we can tackle verification problems that are beyond reach of the proof tools currently available for SPARK. To demonstrate that our methodology is suitable for real-world applications, we show how it can be used to verify an efficient library for big numbers. This library is then used as a basis for an implementation of the RSA public-key encryption algorithm in SPARK/Ada.

1 Introduction

Software for security-critical applications, such as a data encryption algorithm in a virtual private network (VPN) gateway, needs to be particularly trustworthy. If the encryption algorithm does not work as specified, data transmitted over the network may be decrypted or manipulated by an adversary. Moreover, flaws in the implementation may also make the VPN gateway vulnerable to overflows, enabling an attacker to obtain access to the system, or cause the whole gateway to crash. If such a gateway is part of the VPN of a bank, implementation flaws can easily cause considerable financial damage. For that reason, there is a strong economic motivation to avoid bugs in software for such application areas.

Since software controls more and more areas of daily life, software bugs have received increasing attention. In 2006, a bug was introduced into the key generation tool of OpenSSL that was part of the Debian distribution. As a consequence of this bug, the random number generator for producing the keys no longer worked properly, making the generated keys easily predictable and therefore insecure [6]. This bug went unnoticed for about two years.

Although it is commonly accepted that the only way to make sure that software conforms to its specification is to *formally prove* its correctness, it was not until recently that verification tools have reached a sufficient level of maturity to be industrially applicable. A prominent example of such a tool is the SPARK system [2]. It is developed by Altran Praxis and is widely used in industry, notably in the area of avionics. SPARK is currently being used to develop the UK's next-generation air traffic control system *iFACTS*, and has already been

[*] Supported by Federal Office for Information Security (BSI) under grant 880

successfully applied to the verification of a biometric software system in the context of the *Tokeneer* project funded by the NSA [3]. The SPARK system analyzes programs written in a subset of the Ada language, and generates logical formulae that need to hold in order for the programs to be correct. Since it is undecidable in general whether a program meets its specification, not all of these generated formulae can be proved automatically. In this paper, we therefore present the HOL-SPARK verification environment that couples the SPARK system with the interactive proof assistant *Isabelle/HOL* [13].

SPARK imposes a number of restrictions on the programmer to ensure that programs are well-structured and thus more easily verifiable. Pointers and GOTOs are banned from SPARK programs, and for each SPARK procedure, the programmer must declare the intended direction of dataflow. This may sound cumbersome, but eventually leads to code of much higher quality. In standard programming languages, requirements on input parameters or promises about output parameters of procedures, also called *pre-* and *postconditions*, such as "`i` must be smaller than the length of the array `A`" or "`x` will always be greater than 1" are usually written as comments in the program, if at all. These comments are not automatically checked, and often they are wrong, for example when a programmer modified a piece of code but forgot to ensure that the comment still reflects the actual behaviour of the code. SPARK allows the programmer to write down pre- and postconditions of a procedure as logical formulae, and a link between these conditions and the code is provided by a formal correctness proof of the procedure, which makes it a lot easier to detect missing requirements. Moreover, the obligation to develop the code in parallel with its specification and correctness proof facilitates the production of code that immediately works as expected, without spending hours on testing and bug fixing. Having a formal correctness proof of a program also makes it easier for the programmer to ensure that changes do not break important properties of the code.

The rest of this paper is structured as follows. In §2, we give some background information about SPARK and our verification tool chain. In §3, we illustrate the use of our verification environment with a small example. As a larger application, we discuss the verification of a big number library in §4. A brief overview of related work is given in §5. Finally, §6 contains an evaluation of our approach and an outlook to possible future work.

2 Basic Concepts

2.1 SPARK

SPARK [2] is a subset of the Ada language that has been designed to allow verification of high-integrity software. It is missing certain features of Ada that can make programs difficult to verify, such as *access types*, *dynamic data structures*, and *recursion*. SPARK allows to prove absence of *runtime exceptions*, as well as *partial correctness* using pre- and postconditions. Loops can be annotated with *invariants*, and each procedure must have a *dataflow annotation*, specifying the dependencies of the output parameters on the input parameters of the procedure. Since SPARK annotations are just written as comments, SPARK programs can be compiled by an ordinary Ada compiler such as GNAT. SPARK comes with a number of tools, notably the *Examiner* that, given a SPARK program as an input, performs a *dataflow analysis* and generates *verification conditions* (VCs) that must be proved in order for the program to be exception-free and partially correct. The VCs generated by the Examiner are formulae expressed in a language called FDL, which is first-order logic extended with arithmetic operators, arrays, records, and enumeration types. For example, the FDL expression

```
for_all(i: integer, ((i >= min) and (i <= max)) ->
  (element(a, [i]) = 0))
```

states that all elements of the array `a` with indices greater or equal to `min` and smaller or equal to `max` are 0. VCs are processed by another SPARK tool called the *Simplifier* that either completely solves VCs or transforms them into simpler, equivalent conditions. The latter VCs can then be processed using another tool called the *Proof Checker*. While the Simplifier tries to prove VCs in a completely automatic way, the Proof Checker requires user interaction, which enables it to prove formulae that are beyond the scope of the Simplifier. The steps that are required to manually prove a VC are recorded in a log file by the Proof Checker. Finally, this log file, together with the output of the other SPARK tools mentioned above, is read by a tool called POGS (**P**roof **O**bli**G**ation **S**ummariser) that produces a table mentioning for each VC the method by which it has been proved. In order to overcome the limitations of FDL and to express complex specifications, SPARK allows the user to declare so-called *proof functions*. The desired properties of such functions are described by postulating a set of rules that can be used by the Simplifier and Proof Checker [2, §11.7]. An obvious drawback of this approach is that incorrect rules can easily introduce inconsistencies.

2.2 HOL-SPARK

The HOL-SPARK verification environment, which is built on top of Isabelle's object logic HOL, is intended as an alternative to the SPARK Proof Checker, and improves on it in a number of ways. HOL-SPARK allows Isabelle to directly parse files generated by the Examiner and Simplifier, and provides a special proof command to conduct proofs of VCs, which can make use of the full power of Isabelle's rich collection of proof methods. Proofs can be conducted using Isabelle's graphical user interface, which makes it easy to navigate through larger proof scripts. Moreover, proof functions can be introduced in a *definitional* way, for example by using Isabelle's package for recursive functions, rather than by just stating their properties as axioms, which avoids introducing inconsistencies.

Figure 1 shows the integration of HOL-SPARK into the tool chain for the verification of SPARK programs. HOL-SPARK processes declarations (`*.fdl`) and rules (`*.rls`) produced by the Examiner, as well as simplified VCs (`*.siv`) produced by the SPARK Simplifier. Alternatively, the original unsimplified VCs (`*.vcg`)

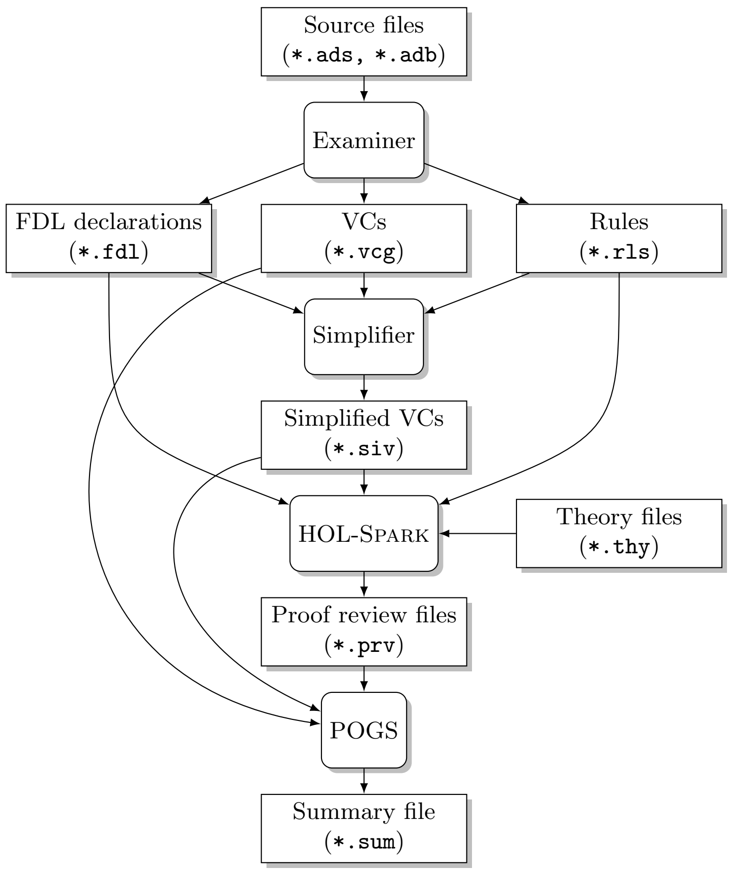

Fig. 1. SPARK program verification tool chain

produced by the Examiner can be used as well. Processing of the SPARK files is triggered by an Isabelle theory file (`*.thy`), which also contains the proofs for the VCs contained in the `*.siv` or `*.vcg` files. Once that all verification conditions have been successfully proved, Isabelle generates a proof review file (`*.prv`) notifying the POGS tool of the VCs that have been discharged.

3 Verifying an Example Program

In this section, we explain the usage of the SPARK verification environment by proving the correctness of an example program for computing the *greatest common divisor* of two natural numbers shown in Fig. 2, which has been taken from the book about SPARK by Barnes [2, §11.6]. In order to specify that the SPARK procedure `G_C_D` behaves like its mathematical counterpart, Barnes introduces a proof function `Gcd` in the package specification.

```
package Greatest_Common_Divisor
is
   --# function Gcd (A, B : Natural) return Natural;

   procedure G_C_D (M, N : in Natural; G : out Natural);
     --# derives G from M, N;
     --# post G = Gcd (M, N);

end Greatest_Common_Divisor;

package body Greatest_Common_Divisor
is
   procedure G_C_D (M, N : in Natural; G : out Natural)
   is
      C, D, R : Natural;
   begin
      C := M; D := N;
      while D /= 0
        --# assert Gcd (C, D) = Gcd (M, N);
      loop
         R := C mod D;
         C := D; D := R;
      end loop;
      G := C;
   end G_C_D;

end Greatest_Common_Divisor;
```

Fig. 2. SPARK program for computing the greatest common divisor

3.1 Importing SPARK VCs into Isabelle

Invoking the Examiner and Simplifier on this program yields a file `g_c_d.siv` containing the simplified VCs, as well as files `g_c_d.fdl` and `g_c_d.rls`, containing FDL declarations and rules, respectively. For `G_C_D` the Examiner generates nine VCs, seven of which are proved automatically by the Simplifier. We now show how to prove the remaining two VCs interactively using HOL-SPARK. For this purpose, we create a *theory* `Greatest_Common_Divisor`, which is shown in Fig. 3. Each proof function occurring in the specification of a SPARK program must be linked with a corresponding Isabelle function. This is accomplished by the command **spark_proof_functions**, which expects a list of equations *name* = *term*, where *name* is the name of the proof function and *term* is the corresponding Isabelle term. In the case of `gcd`, both the SPARK proof function and its Isabelle counterpart happen to have the same name. Isabelle checks that the type of the term linked with a proof function matches the type of the function declared in the `*.fdl` file. We now instruct Isabelle to open a new verification environment and load a set of VCs. This is done using the command **spark_open**, which

```
theory Greatest_Common_Divisor
imports SPARK GCD
begin

spark_proof_functions
  gcd = "gcd :: int ⇒ int ⇒ int"

spark_open "out/greatest_common_divisor/g_c_d.siv"

spark_vc procedure_g_c_d_4
  using ‘0 < d‘ ‘gcd c d = gcd m n‘
  by (simp add: gcd_non_0_int)

spark_vc procedure_g_c_d_9
  using ‘0 ≤ c‘ ‘gcd c 0 = gcd m n‘
  by simp

spark_end

end
```

Fig. 3. Correctness proof for the greatest common divisor program

must be given the name of a `*.siv` or `*.vcg` file as an argument. Behind the scenes, Isabelle parses this file and the corresponding `*.fdl` and `*.rls` files, and converts the VCs to Isabelle terms.

3.2 Proving the VCs

The two open VCs are `procedure_g_c_d_4` and `procedure_g_c_d_9`, both of which contain the `gcd` proof function that the Simplifier does not know anything about. The proof of a particular VC can be started with the **spark_vc** command. The VC `procedure_g_c_d_4` requires us to prove that the `gcd` of `d` and the remainder of `c` and `d` is equal to the `gcd` of the original input values `m` and `n`, which is the *invariant* of the procedure. This is a consequence of the following theorem

```
0 < y ⟹ gcd x y = gcd y (x mod y)
```

The VC `procedure_g_c_d_9` says that if the loop invariant holds when we exit the loop, which means that `d = 0`, then the postcondition of the procedure will hold as well. To prove this, we observe that `gcd c 0 = c` for non-negative `c`. This concludes the proofs of the open VCs, and hence the SPARK verification environment can be closed using the command **spark_end**. This command checks that all VCs have been proved and issues an error message otherwise. Moreover, Isabelle checks that there is no open SPARK verification environment when the final **end** command of a theory is encountered.

4 A verified big number library

We will now apply the HOL-Spark environment to the verification of a library for big numbers. Libraries of this kind form an indispensable basis of algorithms for public key cryptography such as RSA or elliptic curves, as implemented in libraries like OpenSSL. Since cryptographic algorithms involve numbers of considerable size, for example 256 bytes in the case of RSA, or 40 bytes in the case of elliptic curves, it is important for arithmetic operations to be performed as efficiently as possible.

4.1 Introduction to modular multiplication

An operation that is central to many cryptographic algorithms is the computation of $x \cdot y \;\mathbf{mod}\; m$, which is called *modular multiplication*. An obvious way of implementing this operation is to apply the standard multiplication algorithm, followed by division. Since division is one of the most complex operations on big numbers, this approach would not only be very difficult to implement and verify, but also computationally expensive. Therefore, big number libraries often use a technique called *Montgomery multiplication* [10, §14.3.2]. We can think of a big number x as an array of words $x_0, \ldots, x_{n-1}$, where $0 \leq x_i$ and $x_i < b$, and

$$x = \sum_{0 \leq i < n} b^i \cdot x_i$$

In implementations, b will usually be a power of 2. For two big numbers x and y, Montgomery multiplication (denoted by $x \otimes y$) yields

$$x \otimes y = x \cdot y \cdot R^{-1} \;\mathbf{mod}\; m$$

where $R = b^n$, and R^{-1} denotes the multiplicative inverse of R modulo m. Now, in order to compute the product of two numbers x and y modulo m, we first compute the *residues* $\widetilde{x}$ and $\widetilde{y}$ of these numbers, where $\widetilde{x} = x \cdot R \;\mathbf{mod}\; m$ and $\widetilde{y}$ likewise. A residue $\widetilde{x}$ can be computed by a Montgomery multiplication of x with $R^2 \;\mathbf{mod}\; m$, since

$$x \otimes (R^2 \;\mathbf{mod}\; m) = x \cdot R^2 \cdot R^{-1} \;\mathbf{mod}\; m = x \cdot R \;\mathbf{mod}\; m$$

We then have that

$$\widetilde{x} \otimes \widetilde{y} = x \cdot R \cdot y \cdot R \cdot R^{-1} \;\mathbf{mod}\; m = x \cdot y \cdot R \;\mathbf{mod}\; m = \widetilde{x \cdot y}$$

The desired result of the modular multiplication can be obtained by performing a Montgomery multiplication of $\widetilde{x \cdot y}$ with 1, since

$$\widetilde{x \cdot y} \otimes 1 = x \cdot y \cdot R \cdot 1 \cdot R^{-1} \;\mathbf{mod}\; m = x \cdot y \;\mathbf{mod}\; m$$

Before we come to the implementation and verification of Montgomery multiplication, we try to give an intuitive explanation of how the algorithm works. Our

```
$a \leftarrow 0$
for $i = n - 1$ downto $0$ do
  $a \leftarrow a \cdot b + x_i \cdot y$
end for
```

	$0 \cdot 10 =$
	$0 +$
$4 \cdot 789$	$3156 =$
	$3156 \cdot 10 =$
	$31560 +$
$5 \cdot 789$	$3945 =$
	$35505 \cdot 10 =$
	$355050 +$
$6 \cdot 789$	$4734 =$
	359784

```
$a \leftarrow 0$
for $i = 0$ to $n - 1$ do
  $a \leftarrow (a + x_i \cdot y)/b$
end for
```

	0	$+$
$6 \cdot 789$	4734	$=$
	4734	$/\ 10 =$
	473.4	$+$
$5 \cdot 789$	3945	$=$
	4418.4	$/\ 10 =$
	441.84	$+$
$4 \cdot 789$	3156	$=$
	3597.84	$/\ 10 =$
	359.784	

Fig. 4. Two variants of multiplication

exposition is inspired by a note due to Kochanski [9]. As a running example, we take $b = 10$ and assume we would like to multiply 456 with 789. Fig. 4 shows two multiplication algorithms in pseudocode notation, and the tables below the algorithms illustrate the computation steps performed by them. The algorithm on the left is the usual "school multiplication": the multiplier x is processed from left to right, i.e. starting with the most significant digit, and the accumulator a is shifted to the left, i.e. multiplied with 10 in each step. In contrast, the algorithm on the right processes the multiplier from right to left, i.e. starting with the least significant digit, and shifts the accumulator to the right, i.e. divides it by 10. Consequently, the algorithm on the right computes $x \cdot y \cdot R^{-1}$ instead of $x \cdot y$. We now explain how the algorithm on the right can be modified to perform modular multiplication. It might seem that the algorithm requires computations involving floating point numbers, since $a + x_i \cdot y$ is not necessarily divisible by b. However, when working modulo m, this can easily be fixed by adding a suitable multiple of m to $a + x_i \cdot y$, which does not change the result modulo m. The factor by which we have to multiply m is $u = (a_0 + x_i \cdot y_0) \cdot m' \mathbf{\ mod\ } b$, where $m' = -m_0^{-1} \mathbf{\ mod\ } b$ is the additive inverse of the multiplicative inverse of m_0 modulo b, i.e. $(1 + m' \cdot m_0) \mathbf{\ mod\ } b = 0$ and $0 \leq m' < b$. The inverse only exists if m_0 and b are *coprime*, i.e. $gcd(m_0, b) = 1$, which is the case in practical applications, since b will usually be a power of 2 and m will be a large prime number. Note that in order to compute u, we only have to consider the least significant words a_0, y_0 and m_0 of the numbers a, y and m, respectively. It is easy to see that $a + x_i \cdot y + u \cdot m$ is divisible by b, since

$$(a + x_i \cdot y + u \cdot m) \mathbf{\ mod\ } b = (a_0 + x_i \cdot y_0 + (a_0 + x_i \cdot y_0) \cdot m' \cdot m_0) \mathbf{\ mod\ } b =$$
$$(a_0 + x_i \cdot y_0) \cdot (1 + m' \cdot m_0) \mathbf{\ mod\ } b = 0$$

Fig. 5 shows the pseudocode for the Montgomery multiplication algorithm, which employs the ideas described above. As for the other algorithms, we also include

```
$a \leftarrow 0$
**for** $i = 0$ **to** $n - 1$ **do**
    $u \leftarrow (a_0 + x_i \cdot y_0) \cdot m' \;\mathbf{mod}\; b$
    $a \leftarrow (a + x_i \cdot y + u \cdot m)/b$
**end for**
**if** $a \geq m$ **then**
    $a \leftarrow a - m$
**end if**
```

	$0 +$
$6 \cdot 789$	$4734 =$
	$4734 +$
$8 \cdot 987$	$7896 =$
	$12630 \ / \ 10 \ =$
	$1263 +$
$5 \cdot 789$	$3945 =$
	$5208 +$
$6 \cdot 987$	$5922 =$
	$11130 \ / \ 10 \ =$
	$1113 +$
$4 \cdot 789$	$3156 =$
	$4269 +$
$3 \cdot 987$	$2961 =$
	$7230 \ / \ 10 \ =$
	723

Fig. 5. Montgomery multiplication algorithm

a table illustrating the computation. We again multiply the numbers 456 and 789, and use 987 as a modulus. Note that $m' = 7$, since $(1 + 7 \cdot 7) \;\mathbf{mod}\; 10 = 0$. The result of the multiplication is easily seen to be correct, since

$$723 \cdot 1000 \;\mathbf{mod}\; 987 = 516 = 456 \cdot 789 \;\mathbf{mod}\; 987$$

After termination of the loop, it may be necessary to subtract m from a, since a may not be smaller than m, although it will always be smaller than $2 \cdot m - 1$.

4.2 Overview of the big number library

In this section, we give an overview of the big number library and its interface. We have chosen to represent big numbers as *unconstrained arrays* of 64-bit words, where the array indices can range over the natural numbers. All procedures in the big number library operate on segments of unconstrained arrays that are selected by specifying the first and last index of the segment. In situations where a procedure operates on several segments, all of which must have the same length, the last index is usually omitted. The prelude of the `Bignum` library containing the basic declarations is shown in Fig. 6. The big number library provides the following operations:

– Basic big number operations: doubling, subtracting, and comparing
– Precomputation of the values $R^2 \;\mathbf{mod}\; m$ and $-m_0^{-1} \;\mathbf{mod}\; b$
– Montgomery multiplication
– Exponentiation using Montgomery multiplication

The value $R^2 \;\mathbf{mod}\; m = \left(\left(2^k\right)^n\right)^2 \;\mathbf{mod}\; m = 2^{2 \cdot k \cdot n} \;\mathbf{mod}\; m$ can be computed by initializing an accumulator with 1 and applying the doubling operation to it $2 \cdot k \cdot n$

```
package Bignum
is
   Word_Size : constant := 64;
   Base : constant := 2 ** Word_Size;
   type Word is mod Base;
   type Big_Int is array (Natural range <>) of Word;

   --# function Num_Of_Big_Int (A: Big_Int; K, I: Natural)
   --#   return Universal_Integer;

   --# function Num_Of_Boolean (B: Boolean)
   --#   return Universal_Integer;

   --# function Inverse (M, X: Universal_Integer)
   --#   return Universal_Integer;
                                              ...
end Bignum;
```

Fig. 6. Prelude of the big number library

times. After each doubling step, we check whether a carry bit was produced or the resulting number is greater or equal to m, in which case we have to subtract m from the current value of the accumulator. The value $-m_0^{-1}$ **mod** b can be computed by a variant of Euclid's algorithm shown in §3.

Since the specification of the big number operations will make use of constructs that cannot be easily expressed with SPARK's annotation laguage, we have to introduce a number of proof functions. First of all, we need a function that abstracts a big number to a number in the mathematical sense. This function, which is called `Num_Of_Big_Int`, takes an array `A`, together with the first index `K` and the length `I` of the segment representing the big number, and returns a result of type `Universal_Integer`. The Isabelle counterpart of this function is

```
num_of_big_int :: (int ⇒ int) ⇒ int ⇒ int ⇒ int
num_of_big_int A k i = (∑j = 0..<i. Base^j * A (k + j))
```

An array with elements of type τ is represented by the function type `int` $\Rightarrow \tau$ in Isabelle. Function `num_of_big_int` enjoys the following summation property

```
num_of_big_int A k (i + j) =
num_of_big_int A k i + Base^i * num_of_big_int A (k + i) j
```

It is important to note that it would not have been adequate to choose `Integer` instead of `Universal_Integer` as a result type, since the former corresponds to *machine integers* limited to a fixed size, whereas the latter corresponds to the *mathematical* ones. When dealing with operations returning *carry bits*, it is often useful to have a function for converting boolean values to numbers, where

```
procedure Mont_Mult
  (A : out Big_Int; A_First : in Natural; A_Last : in Natural;
   X : in Big_Int; X_First : in Natural;
   Y : in Big_Int; Y_First : in Natural;
   M : in Big_Int; M_First : in Natural;
   M_Inv : in Word);
--# derives
--#   A from
--#   A_First, A_Last, X, X_First, Y, Y_First, M, M_First, M_Inv;
--# pre
--#   A_First in A'Range and A_Last in A'Range and
--#   A_First < A_Last and
--#   X_First in X'Range and
--#   X_First + (A_Last - A_First) in X'Range and
--#   ...
--#   Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) <
--#   Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
--#   1 < Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
--#   1 + M_Inv * M (M_First) = 0;
--# post
--#   Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) =
--#   (Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) *
--#    Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) *
--#    Inverse (Num_Of_Big_Int (M, M_First, A_Last - A_First + 1),
--#      Base) ** (A_Last - A_First + 1)) mod
--#   Num_Of_Big_Int (M, M_First, A_Last - A_First + 1);
```

Fig. 7. Specification of Montgomery multiplication

False and True are converted to 0 and 1, respectively. This is accomplished by the proof function Num_Of_Boolean. Finally, for writing down the specification of Montgomery multiplication, we also need the proof function Inverse denoting the multiplicative inverse of X modulo M. It corresponds to the Isabelle function minv::int ⇒ int ⇒ int, which has the following central property

```
coprime x m ⟹ 0 < x ⟹ 1 < m ⟹ x * minv m x mod m = 1
```

Moreover, if n' is the multiplicative inverse of n modulo m, multiplying k by n' is equivalent modulo m to dividing k by n, provided that k is divisible by n:

```
n * n' mod m = 1 ⟹ k mod n = 0 ⟹ k div n mod m = k * n' mod m
```

This property does not hold if k mod n $\neq$ 0. For example, 5 * 13 mod 16 = 1 and 10 * 13 mod 16 = 2 = 10 div 5, but 9 * 13 mod 16 = 5 $\neq$ 1 = 9 div 5.

4.3 Montgomery multiplication

The central operation in the big number library is Montgomery multiplication, whose specification is shown in Fig. 7. It multiplies `X` with `Y` and stores the result in `A`. The precondition requires the second factor `Y` to be smaller than the modulus `M`. Due to the construction of the algorithm, the first factor `X` is not required to be smaller than `M` in order for the result to be correct. For technical reasons, `A_Last` must be greater than `A_First`, i.e. the length of the big number must be at least 2. This is not a serious restriction, since big numbers of length 1 would be rather pointless. Moreover, the modulus is required to be greater than 1. The precondition *`1 + M_Inv * M (M_First) = 0`* states that `M_Inv` must be the additive inverse of the multiplicative inverse modulo b of the least significant word of the modulus. The postcondition essentially states that $a = x \cdot y \cdot (b^{-1})^n \,\mathbf{mod}\, m$, where n is the length of the big numbers involved, and a, x, y, m are the numbers represented by the arrays `A`, `X`, `Y`, `M`, respectively.

We are now ready to describe the implementation of Montgomery multiplication, which is shown in Fig. 8. Recall that in each step of the Montgomery multiplication algorithm outlined in §4.1, we have to compute $(a + x_i \cdot y + u \cdot m)/b$, where x_i and u are words, and a, y and m are big numbers. In our code for computing this value, we use an optimization technique suggested by Myreen [12, §3.2], which he used for the verification of an ARM machine code implementation of Montgomery multiplication in HOL4. The idea is to perform the two multiplications of a word with a big number, as well as the two addition operations in one single loop. The computation will be done in-place, meaning that the old value of a will be overwritten with the new value. Moreover, since $a + x_i \cdot y + u \cdot m$ is divisible by b, we also shift the array containing the result by one word to the left while performing the computation, which corresponds to a division by b. This is accomplished by the procedure `Add_Mult_Mult` with postcondition

```
Num_Of_Big_Int (A~, A_First + 1, A_Last - A_First + 1) +
Num_Of_Big_int (Y, Y_First, A_Last - A_First + 1) * XI +
Num_Of_Big_int (M, M_First, A_Last - A_First + 1) * U +
Carry1~ + Base * Carry2~ =
Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
Base ** (A_Last - A_First + 1) * (Carry1 + Base * Carry2)
```

The array representing $(a + x_i \cdot y + u \cdot m)/b$ needs to be one word longer than the length of y and m, although the final result of Montgomery multiplication will have the same length as the input numbers. We therefore store the most significant word of a in a separate variable `A_MSW` that is discarded at the end of the computation. To simplify the implementation of the computation described above, we first implement an auxiliary procedure `Single_Add_Mult_Mult` for computing $a_j + x_i \cdot y_j + u \cdot m_j$, where all the operands involved are words. Procedure `Add_Mult_Mult` just iteratively applies this auxiliary procedure to the elements of the big numbers involved.

The **assert** annotation after the **for** command in Fig. 8 specifies the loop invariant, which is

```
procedure Mont_Mult
  ...
is
   Carry : Boolean;
   Carry1, Carry2, A_MSW, XI, U : Word;
begin
   Initialize (A, A_First, A_Last); A_MSW := 0;

   for I in Natural range A_First .. A_Last
     --# assert ...
   loop
      Carry1 := 0; Carry2 := 0;
      XI := X (X_First + (I - A_First));
      U := (A (A_First) + XI * Y (Y_First)) * M_Inv;
      Single_Add_Mult_Mult
        (A (A_First), XI, Y (Y_First),
         M (M_First), U, Carry1, Carry2);
      Add_Mult_Mult
        (A, A_First, A_Last - 1,
         Y, Y_First + 1, M, M_First + 1,
         XI, U, Carry1, Carry2);
      A (A_Last) := A_MSW + Carry1;
      A_MSW := Carry2 + Word_Of_Boolean (A (A_Last) < Carry1);
   end loop;

   if A_MSW /= 0 or else
     not Less (A, A_First, A_Last, M, M_First) then
      Sub_Inplace (A, A_First, A_Last, M, M_First, Carry);
   end if;
end Mont_Mult;
```

Fig. 8. Implementation of Montgomery multiplication

```
(Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
 Base ** (A_Last - A_First + 1) * A_MSW) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) =
(Num_Of_Big_Int (X, X_First, I - A_First) *
 Num_Of_Big_Int (Y, Y_First, A_Last - A_First + 1) *
 Inverse (Num_Of_Big_Int (M, M_First, A_Last - A_First + 1),
   Base) ** (I - A_First)) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
Num_Of_Big_Int (A, A_First, A_Last - A_First + 1) +
Base ** (A_Last - A_First + 1) * A_MSW <
2 * Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) - 1
```

Using a more compact mathematical notation, this invariant can be written as

$$a \mathbf{\ mod\ } m = (x|_j \cdot y \cdot b^{-j}) \mathbf{\ mod\ } m \wedge a < 2 \cdot m - 1$$

where $x|_j$ denotes the number represented by the segment of the array `X` of length $j = \texttt{I} - \texttt{A\_First}$ starting at index `X_First`. The result a computed by the loop can be greater or equal to the modulus, in which case we have to subtract the modulus `M` in order to get the desired result. If $\texttt{A\_MSW} \neq 0$, this obviously means that $m < a$. If $\texttt{A\_MSW} = 0$, we have to check whether $m \leq a$ Since $a < 2 \cdot m - 1$, it suffices to subtract the modulus at most once [10, §14.3.2].

5 Related Work

The design of HOL-SPARK is heavily inspired by the HOL-Boogie environment by Böhme et al. [4] that links Isabelle with Microsoft's Verifying C Compiler (VCC) [5]. The *Victor* tool by Jackson [8], which is distributed with the latest SPARK release, uses a different approach. Victor is a command-line tool that can parse files produced by the SPARK tools, and can transform them into a variety of formats, notably input files for SMT-solvers. Victor has recently been extended to produce Isabelle theory files as well. The drawback of using Victor in connection with Isabelle is that theory files have to be regenerated whenever there is a change in the files produced by SPARK. This can happen quite frequently in the development phase, for example when the user notices that some loop invariant has to be strengthened, or the code has to be restructured in order to simplify verification. The Frama-C system and its *Jessie* plugin [11] for the verification of C code can generate VCs for a number of automatic and interactive provers, including Coq and Isabelle.

A similar big number library written in a C-like language has been proved correct in Isabelle/HOL by Fischer [7] using a verification environment due to Schirmer [14]. This library also includes division, but no Montgomery multiplication. Due to the use of linked lists with pointers instead of arrays, Fischer's formalization is a bit more complicated than ours. Apart from Myreen's work mentioned above, an implementation of Montgomery multiplication in MIPS assembly has been formalized using Coq by Affeldt and Marti [1].

6 Conclusion

We have developed a verification environment for SPARK, which is already part of the Isabelle 2011 release, and have applied it to the verification of a big number library. Our implementation of RSA based on this library reaches about 40% of the speed of OpenSSL when compiled with the `-O3` option on a 64-bit platform. This is quite acceptable, given that OpenSSL uses highly-optimized and hand-written assembly code. A further performance gain could be achieved by using a sliding window exponentiation algorithm instead of the simpler square-and-multiply technique. The library has 743 LOCs, 316 of which (i.e. 43%) are SPARK annotations. The length of the Isabelle files containing correctness proofs of all procedures in the library, as well as necessary background theory, is 1753 lines, of which 391 lines are taken up by the correctness proof for Montgomery multiplication. Development of the library, including proofs, took about three weeks. In the future, we plan to use the library as a basis for an implementation of elliptic curve cryptography. A more long-term goal is to embed the SPARK semantics into Isabelle, to further increase the trustworthiness of VC generation.

Acknowledgement I would like to thank Magnus Myreen for sharing the HOL4 proof scripts of his formalization of Montgomery multiplication with me. Sascha Böhme, Robert Dorn and Alexander Senier commented on draft versions of this paper and helped with optimizations and performance measurements.

References

1. R. Affeldt and N. Marti. An approach to formal verification of arithmetic functions in assembly. In M. Okada and I. Satoh, editors, *11th Annual Asian Computing Science Conf. 2006*, volume 4435 of *LNCS*, pages 346–360. Springer, 2008.
2. J. Barnes. *The SPARK Approach to Safety and Security*. Addison-Wesley, 2006.
3. J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the tokeneer enclave protection software. In A. Hall and J. Wing, editors, *1st International Symposium on Secure Software Engineering*. IEEE, 2006.
4. S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, Feb. 2010.
5. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
6. Debian Security Advisory. DSA-1571-1 OpenSSL – predictable random number generator. Available online at http://www.debian.org/security/2008/dsa-1571.
7. S. Fischer. Formal verification of a big integer library written in C0. Master's thesis, Saarland University, 2006.
8. P. B. Jackson and G. O. Passmore. Proving SPARK Verification Conditions with SMT solvers, 2009.
9. M. Kochanski. Montgomery multiplication: a surreal technique. Available online at http://www.nugae.com/encryption/fap4/montgomery.htm.
10. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.
11. Y. Moy and C. Marché. Jessie plugin tutorial. Technical report, INRIA, 2010. http://frama-c.com/jessie.html.
12. M. O. Myreen and M. J. C. Gordon. Verification of machine code implementations of arithmetic functions for cryptography. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: Emerging Trends Proceedings*. Dept. of Computer Science, University of Kaiserslautern, August 2007. Tech. report 364/07.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
14. N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452, pages 398–414, 2005.

A RSA Encryption / Decryption

The implementation of the RSA encryption and decryption algorithm is shown in Fig. 9. The procedure `Crypt` computes $c = m^e \,\mathbf{mod}\, n$, where c, m, e and n are the numbers represented by the arrays `C`, `M`, `E` and `N`, respectively. When used for encryption, c is the *ciphertext*, m the *plaintext message*, e the *public exponent*, and n the *modulus*, where n is the product of two prime numbers p and q, and $e \cdot d \,\mathbf{mod}\, ((p-1)\cdot(q-1)) = 1$. The same procedure can be used to compute the plaintext from an encrypted message, i.e. $m = c^d \,\mathbf{mod}\, n$. Before calling the Montgomery exponentiation algorithm explained in Appendix B, the procedure precomputes the values $R^2 \,\mathbf{mod}\, n$ and $-n_0^{-1} \,\mathbf{mod}\, b$. Since the exponentiation algorithm requires several auxiliary arrays for storing intermediate results of the computation, we define an array type of fixed length, which will be used for the message `M`, the modulus `N` and the ciphertext `C`:

```
subtype Mod_Range is Natural range 0 .. 63;
subtype Mod_Type is Bignum.Big_Int (Mod_Range);
```

This allows the `Crypt` function to allocate memory for the auxiliary arrays, rather than requiring the caller of `Crypt` to pass suitable arrays as arguments. We have set the length of `Mod_Type` to 64, meaning that it can contain values with $64 \cdot 64 = 4096$ bits, which is sufficient for most practical applications. However, the algorithm and its correctness proof would work equally well for different lengths of `Mod_Type`. Note that the length of the exponent `E` is still unconstrained and need not be the same as the length of the modulus. Indeed, it is quite common to choose public and private exponents that have a different length.

B Exponentiation

The implementation of exponentiation using Montgomery multiplication is shown in Fig. 10. This procedure computes the result $a = x^e \,\mathbf{mod}\, m$, where a, x, e and m are the numbers represented by the arrays `A`, `X`, `E` and `M`, respectively. The algorithm needs a number of auxiliary variables to store intermediate values. These intermediate values are big numbers whose size is not known at compile time, but depends on the size of the unconstrained arrays passed as arguments to the procedure. Since SPARK does not allow the dynamic allocation of memory for data structures, these auxiliary variables need to be created by the caller, and passed to the procedure as arguments, too. This is why `Mont_Exp` has the extra arguments `Aux1`, `Aux2`, and `Aux3`. The parameter `RR` must contain the big number $R^2 \,\mathbf{mod}\, m$, and $1 + \texttt{M\_Inv} \cdot m_0 \,\mathbf{mod}\, b = 0$. We start by initializing `Aux1` with the big number 1. The variable `Aux3`, which we use as an accumulator for computing the result, is set to $\widetilde{1} = R \,\mathbf{mod}\, m$ using `Mont_Mult` (see §4.1). Moreover, we store $\widetilde{x}$ in `Aux2`. The algorithm uses the square-and-multiply approach. It processes the exponent from the most significant bit to the least significant bit. In each iteration `Aux3` is squared, and the result stored in `A`. If the current

```
procedure Crypt
  (E : in Bignum.Big_Int;
   N : in Mod_Type;
   M : in Mod_Type;
   C : out Mod_Type)
is
   Aux1, Aux2, Aux3, RR : Mod_Type;
   N_Inv : Types.Word32;
begin
   Bignum.Size_Square_Mod
     (N, N'First, N'Last, RR, RR'First);

   N_Inv := Bignum.Word_Inverse (N (N'First));

   Bignum.Mont_Exp
     (C, C'First, C'Last,
      M, M'First,
      E, E'First, E'Last,
      N, N'First,
      Aux1, Aux1'First,
      Aux2, Aux2'First,
      Aux3, Aux3'First,
      RR, RR'First,
      N_Inv);
end Crypt;
```

Fig. 9. Implementation of RSA algorithm

bit of the exponent is set, `A` is multiplied with `Aux2` (containing $\widetilde{x}$), and the result is stored in `Aux3` again, otherwise `A` is just copied back to `Aux3`. The invariant of the inner loop is

```
Num_Of_Big_Int (Aux1, Aux1_First, A_Last - A_First + 1) = 1 and
Num_Of_Big_Int (Aux2, Aux2_First, A_Last - A_First + 1) =
Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) *
Base ** (A_Last - A_First + 1) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1) and
Num_Of_Big_Int (Aux3, Aux3_First, A_Last - A_First + 1) =
Num_Of_Big_Int (X, X_First, A_Last - A_First + 1) **
(Num_Of_Big_Int (E, I + 1, E_Last - I) * 2 ** (Word_Size - 1 - J) +
 Universal_Integer (E (I)) / 2 ** (J + 1)) *
Base ** (A_Last - A_First + 1) mod
Num_Of_Big_Int (M, M_First, A_Last - A_First + 1)
```

After termination of the loop, `Aux3` is converted from “Montgomery format” to the “normal format” again by Montgomery-multiplying it with 1 and storing the result in `A`.

```
procedure Mont_Exp
  (A : out Big_Int; A_First : in Natural; A_Last : in Natural;
   X : in Big_Int; X_First : in Natural;
   E : in Big_Int; E_First : in Natural; E_Last : in Natural;
   M : in Big_Int; M_First : in Natural;
   Aux1 : out Big_Int; Aux1_First : in Natural;
   ...
   RR : in Big_Int; RR_First : in Natural;
   M_Inv : in Word)
is
begin
   Initialize (Aux1, Aux1_First, Aux1_First + (A_Last - A_First));
   Aux1 (Aux1_First) := 1;

   Mont_Mult
     (Aux3, Aux3_First, Aux3_First + (A_Last - A_First),
      RR, RR_First, Aux1, Aux1_First, M, M_First, M_Inv);

   Mont_Mult
     (Aux2, Aux2_First, Aux2_First + (A_Last - A_First),
      X, X_First, RR, RR_First, M, M_First, M_Inv);

   for I in reverse Natural range E_First .. E_Last
   loop
      for J in reverse Natural range 0 .. Word_Size - 1
        --# assert ...
      loop
         Mont_Mult
           (A, A_First, A_Last,
            Aux3, Aux3_First, Aux3, Aux3_First,
            M, M_First, M_Inv);

         if (E (I) and 2 ** J) /= 0 then
            Mont_Mult
              (Aux3, Aux3_First, Aux3_First + (A_Last - A_First),
               A, A_First, Aux2, Aux2_First,
               M, M_First, M_Inv);
         else
            Copy (A, A_First, A_Last, Aux3, Aux3_First);
         end if;
      end loop;
   end loop;

   Mont_Mult
     (A, A_First, A_Last,
      Aux3, Aux3_First, Aux1, Aux1_First, M, M_First, M_Inv);
end Mont_Exp;
```

Fig. 10. Implementation of exponentiation

Verification of Safety-Critical Systems: A Case Study Report on Using Modern Model Checking Tools

Antti Jääskeläinen[1], Mika Katara[1], Shmuel Katz[2], and Heikki Virtanen[1]

[1] Department of Software Systems, Tampere University of Technology
PO BOX 553, 33101 Tampere, Finland
Tel. +358 40 849 0743, Fax. +358 3 3115 2913
`{antti.m.jaaskelainen,mika.katara,heikki.virtanen}@tut.fi`
[2] Department of Computer Science
Technion – Israel Institute of Technology
`katz@cs.technion.ac.il`

Abstract. Formal methods are making their way into the development of safety-critical systems. In this paper, we describe a case study where a simple 2oo3 voting scheme for a shutdown system was verified using two bounded model checking tools, CBMC and EBMC. The system represents Systematic Capability level 3 according to IEC 61508 ed2.0. The verification process was based on requirements and pseudo code, and involved verifying C and Verilog code implementing the pseudo code. The results suggest that the tools were suitable for the task, but require considerable training to reach productive use for code embedded in industrial equipment. We also identified some issues in the development process that could be streamlined with the use of more formal verification methods. Towards the end of the paper, we discuss the issues we found and how to address them in a practical setting.

1 Introduction

Companies developing safety-critical systems must balance between safety requirements imposed by standards and productivity requirements. On the one hand, the higher the safety integrity requirements, the more time and effort are needed for validation and verification activities. On the other hand, companies producing less safety-critical systems often face fierce competition and are required to put more emphasis on the overall efficiency of the development process.

Certification is another driving force in the field. Many companies are trying to get their products certified in order to help marketing efforts. The new machinery directive in the EU, for instance, is still based on self-declaration in the case of most type of machines; the manufacturer labels the product with the "CE" marking without formal type examination. However, certification by an independent assessment organization may still be required by customers and/or for marketing reasons. It is also seen as an important step if an accident should occur and investigation of the development practices takes place.

IEC 61508 [15] is a basic standard on functional safety; a new edition 2 of the standard was released in April 2010. The standard classifies safety-critical systems into four Safety Integrity Levels (SILs), SIL 4 corresponding to the most critical and SIL 1 the least critical type of system. The standard presents methods used for the verification and validation of safety-critical hardware and software. For each SIL level, there is a set of Highly Recommended, Recommended and Not Recommended methods. In addition, for the use of some methods the standard does not indicate any recommendation on certain SIL levels.

Systems can be composed of elements and subsystems having a predetermined Systematic Capability (SC) on the scale 1-4 corresponding to the SIL level of the whole system. For example, SIL 3 level systems can be composed of elements having SC 3 or 4 when used according to the instructions given in the elements' safety manuals. IEC 61508 is not harmonized, i.e. it does not fulfill the requirements of the European directives as such, but is often referred to by other, harmonized, standards (such as EN ISO 13849-1 and EN 62061) in relation to requirements imposed on the development of safety-critical systems.

Formal methods are considered an important technology in the development of safety-critical systems. While the scalability and usability of tools still pose challenges in the development of non-safety-critical systems, safety-critical systems are somewhat different in this respect. On higher SIL/SC levels there are fewer productivity constraints and perhaps more time to learn new techniques that can help in validation and verification efforts. Moreover, safety-critical systems should be kept rather simple in order to limit the needed verification and validation activities. Thus, in spite of the scalability problems, it it often feasible to prove correct at least some parts of the system using formal methods. Moreover, formal methods are well represented in the IEC 61508 standard for developing high SIL level systems. They can also be used on lower SIL levels to replace some less formal techniques, such as certain types of testing.

Nevertheless, there have been major impediments in using formal methods. Performance of the old tools and the computing power available was too limited in order to solve real life problems. Moreover, special expertise was required to use the tools. Nowadays, there is evidence in the literature that new tools can solve practical problems given the increased computing resources available. Unfortunately, however, there is still lack of user experience reports that would discuss the required expertise to use the modern tools.

Towards these ends, we describe a case study where we experimented with a formal verification technology in an industrial case study. The case study subject was a simple 2oo3 (2-out-of-3) voting scheme used for redundancy in a SC 3 level shutdown system. The system development is being done according to the IEC 61508 standard and certification is being conducted by an independent organization.

In the case study we treat a typical industrial development where pseudo-code (or programming language code in, e.g., C) is first written and shown correct relative to requirements of the module under development. This is then handed off to a separate development team as the basis of a hardware or firmware design in a low-level hardware design language such as Verilog or VHDL. This too must be shown correct relative to (often more detailed) requirements. The main practical tasks are (1) to ensure that the high- and low-level properties checked actually express the needed requirements and are easy to write, (2) to facilitate verifying the pseudo-code level relative to the properties, and (3) to similarly show that the chip design satisfies the needed properties, while showing consistency with the upper level solution.

For verification we used two bounded model checking tools, CBMC and EBMC [6]. Model checking as a technology does not require as high a level of expertise as, for instance, theorem proving. Moreover, these tools were easily available and supported the input formats we were able to work with. In addition, they support the existing development process and no major changes in the work flow are required.

While the standard does not require the use of formal verification in the case of this particular system, formal verification can complement less formal verification methods, such as testing and simulation, and somewhat ease the certification process. Moreover, if the development process could be changed in the future to better take advantage of the formal verification technology, some of the less formal techniques could possibly be replaced with it.

Since the verified system is very simple, the focus of this paper is on reporting experiences in using the verification tools in the particular industrial context of safety-critical systems development rather than in the verification technology itself. The results of the case study suggest that while suitable tools might be hard to find, together with the process changes, they could provide better evidence for the correctness of the system. Should it be possible to replace some informal techniques with more formal ones, productivity gains could also be achieved. Nevertheless, the efficient use of model checking tools requires expertise, so considerable training may be needed in order to equip the developers with the skills necessary to use such tools.

The structure of the paper is as follows: In Section 2 we present the background of the tools used in the case study and discuss related work on how formal methods and related tools are used in various tasks in software and hardware development. Section 3 introduces the case study. Due to confidentiality restrictions, some details of the shutdown system have been omitted. Finally, the lessons learned from the case study are discussed in Section 4.

2 Model Checking Safety-Critical Systems

In this section, we first introduce the basic concepts related to model checking in general and bounded model checking in particular. Then we move on to discuss related work on how formal methods and related tools are used in various tasks in software and hardware development.

2.1 Model Checking and Bounded Model Checking

Model checking [8] is a formal verification technique in which all possible execution paths of a model of a system or component are checked for a given property, where the model must have a finite number of possible states (although there can be infinite computations). The property to be checked is generally given in some form of temporal logic [18]. This allows expressing assertions about the final values of a module, invariants that should be true also at intermediate stages, as well as assertions about, e.g., responsiveness of a system to requests or stimuli. The model of the system is called a Kripke structure, and is a graph with nodes that each represent a state of the system, and directed edges where each represents an operation that moves the system from the source state to the target.

The main problem with model checking is that the number of states in a system can become unmanageably large. Thus model checking techniques are intended to overcome this difficulty. Among the classic approaches are representing the states symbolically in data structures known as binary decision diagrams (BDD's), and creating smaller "abstract" models that combine many states into one (so that if the smaller model is shown correct for a desired property, the original model is also guaranteed to be correct).

Model checking tools originally had their own notations for expressing the models, e.g., in the SMV model checking tool [7]. The tool and its notation were used either to show that a design of a key algorithm was correct, or code was translated to the notation of the tool involved [11].

More recently, tools have been developed to directly take as input the code of the component to be checked (e.g., in C or Java), and to use *assert* statements to indicate at what point an assertion should be correct. In addition, the underlying technology of model checkers has changed: today it is common to translate both the model (or code) and the assertions to a complex boolean formula, and use a SAT (satisfiability) solver [19] or extended techniques called SMT [20] to determine whether the formula can be made true for some assignment of values to the variables in it. In fact, the formula constructed is equivalent to encoding the execution of the model, and asserting the *negation* of the property we want. Thus finding a set of values for the variables in the formula is equivalent to finding a counterexample for the property, because it represents a computation of the system that does not satisfy the desired property.

Both in order to create smaller models, and to ensure that any counterexample execution paths are as short as possible, *bounded* model checking has been used. In this approach, a bound is put on the length (number of states) of paths that will be checked. Thus for some n, all possible paths of length up to n are checked. If a counterexample is found, it can be analyzed to detect the bug. While if none exists for paths up to n, the bound can be increased, until a bound longer that any path in the program is reached, or the user decides that longer paths can be ignored.

Modules to ensure safety-critical properties of industrial software often regulate control or repeatedly test whether shut-down is necessary. Such modules are generally limited in their state-space, and each round of application is bounded in length. Thus bounded model checking is appropriate, and often can achieve full verification. Full model checkers, such as SATabs are appropriate for larger programs, but, as noted on the home webpage of that tool [23], can only automatically check for restricted properties such as array bounds, buffer overflows, or built-in exceptions, because of the needed abstraction step in going from code to a model.

In this work we show a case study where the computations are of a fixed length at each activation of the module investigated, so many of the more complex issues are irrelevant. We investigate whether tools for bounded model checking are sufficiently robust and user-friendly to be practically used to verify and increase the reliability of software or firmware embedded in industrial equipment.

In this case study, we used two bounded model checkers, namely CBMC and EBMC [6]. The former enables software model checking and supports ANSI-C and C++ as input languages. The tool performs verification by first unwinding the code loops and then passing the results in an equational form to a decision procedure (e.g., a SAT solver). In many cases, the tool can check that enough unwinding is performed, and thus the complete state space is considered in the analysis (sound verification). If the formula that encodes the program unwindings is satisfiable, i.e., contains an invalid program path, then the tool will produce a counterexample. There are also command-line options to limit the number of times the loops are unwound or the number of program steps to be processed, and to stop checking that enough unwinding is done; this allows using the tool for bug hunting in cases where no useful bound exists and properties cannot be proven correct. On the other hand, EBMC is a tool for hardware verification supporting input in Verilog and related formats. However, VHDL is not among the supported input formats. Both tools are available in binary format for Windows, Linux and MacOS.

2.2 Related Work

In the following, we describe some application examples of formal verification techniques in relation to software and hardware development. It is worth paying attention to the way model checking is used and what kind of impact it has for the development process and overall quality.

Björkman et al. [4] verified stepwise shutdown logic in the nuclear domain and used model checking in the traditional way: the design was converted to a dedicated verification model and the requirements in the specification were translated into logical formulae. They used a model checking tool for proving that the verification model satisfies the formulae. Obviously, this use of model checking is rather demanding and laborious because of the model transformations needed, but it has some advantages as well. Already while constructing formal models, many omissions and contradictions become clearly visible, and larger systems can be verified because irrelevant details can be omitted from the abstract verification model.

The cited experiment shows the most valuable benefit of formal methods too. Because all of the modeled behavior is fully covered, no issue can hide itself in the verification model. However, the proof is valid only if the abstract verification model corresponds to the design and the formulae cover all of the requirements. One of today's research challenge is to find new ways of applying formal methods so that the artifacts used in proofs would be more closely related to the specification, design and implementation languages used in mainstream software and hardware development; this would reduce the need for error-prone manual transformations.

Even though formal methods may not be applicable always as such, they can still be helpful. For example, testing can benefit from their use. Angeletti et al. [1] reported an experiment in the railway domain where bounded model checking was used to semi-automatically generate test cases in order to gain full coverage requested by the EN 50128 guidelines for the software development of safety-critical systems at SIL 4 level.

In the experiment, the C code was augmented with failing assertions and the CBMC tool was used to compute the values of input parameters for each assertion to be reached. Obviously, the mere values of input parameters are not enough for defining test cases; in order to be useful, the test case must contain checks against the expected outputs. In our case study, such checks were encoded directly as conditions in assertions and verified on the fly. Unfortunately, the paper by Angeletti et al. does not state directly how the expected values were obtained and why on-the-fly verification was not used. A system of a few thousand lines of C code may be too large to be model checked, so the approach we used in our case study may not have been applicable as such, and unlike in our approach, test cases can be used to verify and validate the SUT in binary form without the source code.

In theory, any model having operational semantics can be verified by means of model checking and state transition systems can be used to model many other aspects of the systems than behavior in normal conditions. For example, there is a special Statecharts variant called Safecharts for modeling safety issues and their relations to functional properties [9].

In Safecharts there are special states for normal and defunct states for the components of the system and transitions between them. Events associated with those transitions model the breakdowns and reparations of the components. When these special states and events are synchronized with the states and actions of the functional layer, the behavior of the system can be modeled and formally verified, not only in normal operation, but in those situations in which parts of the system do not function properly [12]. This is very useful if the system cannot reach a safe stable state without controlled operations. In aerospace and nuclear domains this requirement is obvious, but also in the case of complex and big machines there might be a need to shutdown slowly to prevent further breakdowns.

In addition to facilitating testing, formal verification can significantly reduce the need for testing. Kaivola et al. [16] used formal verification as the primary validation vehicle for the execution cluster of the Intel Core i7 processor and dropped most of the usual RTL (Register Transfer Language) simulations and all coverage driven simulation validation. They concluded that verification required approximately the same amount of work as traditional pre-silicon testing. Although not zero, the number of bugs that escaped to silicon was lower than for any other cluster.

In addition to describing how formal verification could replace testing, Kaivola et al. sketch some prerequisites for verification to be applicable in practice. In contexts where model checking can replace simulation-based testing, it can be seen as a clever and effective way of conducting exhaustive simulation.

Nevertheless, even a company like Intel has taken quite some time to introduce formal verification into the development process. Most likely the story began in 1994 when the Pentium FDIV bug was found [22] and seven years later they reported that they had verified the Pentium 4 floating-point divider [17]. As a pioneer in the field, Intel has made enormous investments in formal verification and for others, effort is likely far more moderate. Still, it may take considerable effort to establish the confidence needed to be able to supercede existing verification methods with more formal ones. However, they can used to complement the existing ones and provide diversity when needed.

The systems in the examples discussed above have very high integrity requirements and two of them are also large from the verification point of view. For example, the execution cluster of i7 is responsible for the functional behavior of all of the more than 2700 distinct microinstructions. The majority of safety-critical systems are much smaller and may not have such high integrity requirements. Nonetheless, formal methods can be a feasible alternative for the quality assurance of those because small verification problems are not as laborious to solve as it is generally thought and even small systems can have peculiar and critical faults, which can be almost impossible to find by other means.

3 Case Study

We now present our case study on using model checking to verify a simple element in a safety-critical system. In more detail, the goal was to use model checking tools to verify the implementation of the 2oo3 voting scheme in a SC 3 level shutdown system. This voting scheme (also known as triple-modular redundancy) is very popular in safety-critical systems because it provides a good compromise between safety and availability. Since availability is an important factor in industrial systems, such compromises are often searched for.

There are three distinct modules which receive the same input (from one, two or three different sensors) and shutdown is started when at least two out of three modules suggest it. In this case, the design follows the idle current design, i.e. the output is active when there is no need to shutdown the system. When at least two out of three inputs are active, the output is also active. If only one or zero inputs are active or the power is lost, the output should indicate a need to start the shutdown procedure. In practice, each input is a Boolean value, one indicating a normal situation and zero indicating the need to shutdown the system. If two or three of the input values equal zero, the voter unit outputs value zero and the shutdown procedure begins. If only one input equals zero, the process can continue (with a possible log message indicating some potential problem in the corresponding module). Thus, the system is able to mask a fault in one of the modules, allowing the system to continue its operation. The interested reader is referred to [24, p. 132] for more elaborate discussion on this voting scheme.

3.1 Working with the Pseudo Code

In this case the development process is such that the basic requirements are refined first and then translated into pseudo code. Typically, the pseudo code is augmented with a short textual description that may specify some basic properties of the solution depicted as pseudo code. The pseudo code is then implemented with a suitable concrete language; VHDL in case a programmable hardware solution is preferred. The tests for the implementation are derived from the requirements, which are managed in a requirements management tool.

The first stage of the case study was to verify the pseudo code. The tool used for formal verification was CBMC (version 3.9) and for that purpose the pseudo code was manually translated into C code. Since the implementation of the voting scheme with Boolean values is very simple, manual translation was considered adequate in this particular case. Moreover, because of the simplicity of the code, it was possible to derive eight test cases (2^3) that covered all possible input and output combinations.

The test cases were encoded as assertions in the C code and verified with the tool. This process also revealed that the property specified in conjunction with the associated pseudo code was somewhat vague and incomplete; the informal description didn't cover all the input/output combinations. We think that this represents a typical case of specifying simple designs: even though the requirements should be explicit and complete, it is very easy to ignore some details since the design is considered obvious.

The C code used with CBMC is listed in Figure 1. There are three parameters, corresponding to three inputs to the system; the `OCHY_Voter_State` variable is the output. The actual voting is implemented in the statement where `OCHY_Voter_State` gets assigned a value. The assertions corresponding to the eight test cases follow the assignment. The structure of the assertions was chosen to support understandability in the absence of the implication operator; another possibility would have been to use not and or operators to substitute for implication (and give the original form with the implication operator in a comment above the assertion, for instance). The current form also shows the locality of assertions in C.

3.2 Working with the VHDL Code

The second stage was to verify the actual implementation of the pseudo code in VHDL. Ideally, the verification tool should accept VHDL as input language, but

```
#include<assert.h>
void foo(int OCHY_comparator_state_ICH1,
         int OCHY_comparator_state_ICH2,
         int OCHY_comparator_state_ICH3) {
int OCHY_Voter_State = 0;

OCHY_Voter_State =
  (OCHY_comparator_state_ICH1 || OCHY_comparator_state_ICH2) &&
  (OCHY_comparator_state_ICH1 || OCHY_comparator_state_ICH3) &&
  (OCHY_comparator_state_ICH2 || OCHY_comparator_state_ICH3);

if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 1);
}
if ((OCHY_comparator_state_ICH1 == 1) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 1)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 1)) { assert(OCHY_Voter_State == 0);
}
if ((OCHY_comparator_state_ICH1 == 0) && (OCHY_comparator_state_ICH2 == 0)
 && (OCHY_comparator_state_ICH3 == 0)) { assert(OCHY_Voter_State == 0);
}}
```

Fig. 1. The C code and the eight assertions verified with CBMC.

for practical reasons we chose EBMC (version 4.1). Since EBMC uses Verilog as its input language, we first translated the VHDL code to Verilog using a VHDL to Verilog RTL translator tool [10] (version 2.0). The verification process was not as straightforward as in the case of the C code. We struggled with the syntax and the use of the tool since the information available with the installation package and on the tool website [6] was more limited than in case of CBMC. A significant practical difference with the tools was that the assertions were considered global in EBMC and local in CBMC. This made the reuse of assertions developed for the C code impossible.

```
always @(posedge clk or posedge rst_n) begin
  if(rst_n == 1'b 0) begin
    voter_state_i <= 1'b 0;
  end else begin
    if((ICH1_comparator_state_och_in == 1'b 1 &&
        ICH2_comparator_state_och_in == 1'b 1) ||
       (ICH1_comparator_state_och_in == 1'b 1 &&
        ICH3_comparator_state_och_in == 1'b 1) ||
       (ICH2_comparator_state_och_in == 1'b 1 &&
        ICH3_comparator_state_och_in == 1'b 1))
    begin
      voter_state_i <= 1'b 1;
    end
    else begin
      voter_state_i <= 1'b 0;
    end
  end
end
```

Fig. 2. The implementation of the voting code after VHDL to Verilog translation.

Figure 2 shows how the voting is implemented in the Verilog code. The *always block* gets executed on the rising edge of either the clock or the reset signal. If the reset is active (zero), then the output is zero. Otherwise the voting occurs. Interestingly, the implementation in VHDL did not directly correspond to the original pseudo code, but had `&&` (and) in the innermost level and `||` (or) in the outermost level.

The code shown is generated by the translator tool from the original VHDL source. In practice, with the active low reset signal, it would make more sense to use a falling edge instead of rising edge to trigger the code block. However, the block gets triggered with the next rising edge of the clock signal in any case, so this did not affect the verification task.

The code shown in Figure 3 shows a part that was added to the Verilog code only for the purposes of verification. There are now three new registers: `voter_state_check_in_pos`, `voter_state_check_in_neg`, and `voter_state_check`. The value one of the first register should imply a voting result one. Correspondingly, the value one of the second register should imply a voting result zero. The value of the third register should always be one if the system is working correctly. Since registers in Verilog have unknown initial values by default, the new registers are assigned initial values in the `initial` block.

Figure 4 shows the actual assertion block that gets triggered similarly to the original voting block. If the reset is not active and at least two of the inputs are one, the first new register gets value one. Correspondingly, if the reset is not active and at least two of the inputs are zero, the second new register gets value one. The third new variable gets assigned a value indicating whether the value

```
reg  voter_state_check_in_pos;
reg  voter_state_check_in_neg;
reg  voter_state_check;

initial begin
  voter_state_check_in_pos = 0;
  voter_state_check_in_neg = 0;
  voter_state_check = 1;
end
```

Fig. 3. The added verification code in Verilog – part 1.

```
always @(posedge clk or posedge rst_n)  begin
  voter_state_check_in_pos <= rst_n & (1'b 0
    | (ICH1_comparator_state_och_in & ICH2_comparator_state_och_in)
    | (ICH1_comparator_state_och_in & ICH3_comparator_state_och_in)
    | (ICH2_comparator_state_och_in & ICH3_comparator_state_och_in)
    );
  voter_state_check_in_neg <= rst_n & (1'b 0
    | (!ICH1_comparator_state_och_in & !ICH2_comparator_state_och_in)
    | (!ICH1_comparator_state_och_in & !ICH3_comparator_state_och_in)
    | (!ICH2_comparator_state_och_in & !ICH3_comparator_state_och_in)
    );
  voter_state_check <= (!voter_state_check_in_pos | voter_state_i) &
                       (!voter_state_check_in_neg | !voter_state_i);
  assert (voter_state_check);
end
```

Fig. 4. The added verification code in Verilog – part 2 (please note the use of bitwise operators).

of the first new register implies the voting result and the value of the second one implies the negation of the voting result.

One should note that the assignments are non-blocking, i.e. the right-hand side of each of the assignments is evaluated first. The assignment to the left-hand side is delayed until all the evaluations have been done.

The structure of the code block enables adding and removing "test cases" (input combinations in the context of the corresponding expected output value) from the statements and the expression `1'b 0` ensures that the assertions work also without any "test cases". We think that this is a robust, easy-to-use and reusable solution, since it allows extending the assertions with new properties incrementally. However, since the code in this case study is simple, the benefits are not so visible here.

The solution can be extended into more complex systems. For each bit of output two new registers and assignments to them are added, as well as corresponding terms to the expression of the final assignment. For each bit of input a new term is added to the relevant bitwise conjunctions of the assigned expressions for each expected output value. The assignment for an expected output value is placed into an always block corresponding to the situations where the value of the output may change in the code to be tested. However, this method is limited to stateless systems; a system with internal memory cannot be handled in such a straightforward manner.

One practical problem related to the inexperience of the person using the bounded model checking tools was that it was seemingly easy to verify properties that did not correspond to the actual requirement. For this reason we used a fault seeding technique where we introduced errors to the properties and checked whether it was possible to verify the erroneous properties. If not, we also checked that the counterexample provided by the tool corresponded to the seeded error. In practice, the errors seeded were more or less random changes made to the properties, i.e. we did not follow any systematic pattern. We think that this is a useful and practical technique for engineers without much experience in using model checking tools since it can be used to help determine whether a specification actually captures the desired intention, as is done with tests of vacuity [3, 2], where it is determined whether a subproperty is actually needed in the specification. This allows, for example, showing that an implication is true "by default" because the left side is always false.

All the assertions shown in the figures were verified with the tools. The bound value we used with EBMC was relative low, though. Once we became familiar with the tool, we noticed that the bound given to the tool as a command line option made a big practical difference. First, in some cases, it was possible to find problems in the assertions only when the value of the bound was high enough. This should be taken into account when using the fault seeding technique. Second, while the execution time of the tool with low bound values was reasonable (bound value 1000 corresponded roughly to 10 seconds in verification time with a regular laptop computer), the execution took much more time with higher bound values due to the state space explosion. We also ran into some warning messaging concerning solver inconsistencies and one segmentation fault. Nevertheless, the tools were considered a good choice for the purposes of this small case study. However, especially the EBMC tool would be much more appealing from the practical point of view if a proper user manual and documentation were available.

Regarding future work, creating the test code as used in the case study can be cumbersome if inputs and outputs are numerous. More complicated inputs and outputs such as integers have to be handled bit by bit, which causes even more work. However, since the test code is very regular, it could be generated automatically with a suitable assisting tool. The registers and assignments can be created based on a list of outputs, with those of more complex types converted to a number of single-bit outputs. The expressions for the expected value assignments can be similarly created based on a listing of input combinations with the corresponding outputs, which may be given for example as a CSV (Comma Separated Values) file. In this way test cases can be converted into assertions in the code with little effort using Excel sheets created by test engineers, for instance.

4 Discussion

Even though our case study was small-scale in terms of the code checked, it helped us to identify some potential problems and partial solutions in the context of using model checking techniques to verify safety-critical systems. In more detail, the analysis of the results of the case study led to the following recommendations.

First, formal verification is seen useful at least in simple cases like the one studied. It was possible to develop a generic assertion mechanism for the code translated from VHDL to Verilog, which should be reusable in the verification of similar designs and further supported by assisting tools. Training would still be needed, though, in order to get engineers to use the tools.

While reusing assertions is seen to be beneficial, understanding how to develop effective assertions would need further training in the next step after basic training, unless this is solved by assisting tools. We think that starting with simple "test cases" before moving towards verifying more complicated properties can help in this process. In addition, we recommend using the fault seeding technique where errors are introduced to the properties for the purposes of checking whether it is possible to verify the erroneous properties; in our case this helped us to catch errors in the assertions.

Second, the tools used in this study worked well, but their scalability is still unknown. It would also be better if the VHDL code could be checked directly without the translation process to Verilog, unless a (certified) translator that could be trusted is found.

Third, the design flow in this particular case could be improved by specifying the properties associated with the requirements more precisely. This would allow detecting errors and inconsistencies already in the requirements capturing phase, as this phase is widely recognized to be critical. In an ideal case, the same properties could be translated into assertions used in the formal verification of the VHDL code. These kinds of properties and assertions could be reused in the case of modifications in a regression testing fashion; they could ease the burden of reverification needed in case of modifications that affect many elements. In addition, there might be some generic high level properties and assertions that could be used by different projects as sanity checks for a set of implementations sharing commonalities.

Fourth, experimenting first with tiny systems is highly recommended. Model checking suffers from the state explosion problem like any other formal verification technique and with bigger systems more expertise is required to specify the system and requirements in a way that can be handled with the computing resources available. Moreover, complex specifications are more error prone to write and harder to check.

One practical problem related to the tools might be to find a suitable formal verification tool. Formal verification tools capable of analyzing VHDL exist, such as [13, 5]. Due to high license costs, however, it might be more economical to buy formal verification as a service (see, for instance [21]), if a company has only a limited need for such a tool. This option would also require less training. Another tool-related issue is certification: in principle, the software tools used in developing safety-critical systems should be certified by independent bodies [14, p. 83]. While certification is commonly used for compilers, we are not aware of any certified formal verification tool; this might become an issue in the future on high SIL/SC levels.

To conclude, the practical case study as well as the review of the related work show that model checking is a useful technique in the development of safety-critical systems. While there still are many problems to be solved, the tools are getting more scalable and user-friendly. In particular, it would be essential to provide tools that can work directly on the pseudo or source code used in the development and that require only basic training to be useful. Moreover, the whole development process could be streamlined with the support of such tools. While the standards regulating the development practices in the safety-critical domain are recommending the use of formal verification tools, the biggest problem seems to be related to training, and methodological introduction into the development process that could be eased with the help of simple assisting tools that, for instance, use input formats familiar to the users.

Acknowledgements Funding from Tekes and the companies participating in the Ohjelmaturva project is gratefully acknowledged. The authors would also like to thank the anonymous reviewers for their helpful comments.

References

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. J. Autom. Reason. 45, 397–414 (December 2010), `http://dx.doi.org/10.1007/s10817-010-9172-3`
2. Ball, T., Kupferman, O.: Vacuity in testing. In: Beckert, B., Hähnle, R. (eds.) Tests and Proofs, pp. 4–17. LNCS 4966, Springer Berlin / Heidelberg (2008)
3. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. Formal Methods in System Design 18, 141–162 (2001)
4. Björkman, K., Frits, J., Valkonen, J., Lahtinen, J., Heljanko, K., Hämäläinen, J.J.: Verification of safety logic designs by model checking. In: Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, NPIC&HMIT (2009)
5. Cadence: Incisive Formal Verifier datasheet. `http://www.cadence.com/rl/Resources/datasheets/IncisiveFV_ds.pdf`, cited March 2011
6. CBMC, EBMC: Homepage. `http://www.cprover.org`, cited March 2011
7. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: CAV'99. pp. 495–499. LNCS 1633, Springer (1999), `http://nusmv.itc.it`

8. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA (1999)
9. Dammag, H., Nissanke, N.: Safecharts for specifying and designing safety critical systems. In: In Symposium on Reliable Distributed Systems. pp. 78–87 (1999)
10. Gonzales, M.: VHDL to Verilog RTL translator v2.0. `http://www.ocean-logic.com/downloads.htm`, cited March 2011
11. Hatcliff, J., Dwyer, M.: Using the Bandera Tool Set to model-check properties of concurrent Java software. In: Larsen, K.G., Nielsen, M. (eds.) Proc. 12th Int. Conf. on Concurrency Theory, CONCUR'01. pp. 39–58. LNCS 2154, Springer-Verlag (2001)
12. Hsiung, P.A., Chen, Y.R., Lin, Y.H.: Model checking safety-critical systems using safecharts. IEEE Transactions on Computers 56, 692–705 (2007)
13. IBM: IBM RuleBase homepage. `http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/`, cited March 2011
14. International Electrotechnical Commission: IEC 61508-7, Functional safety of electrical/electronic/programmable electronic safety-related systems, part 7 (2010), ed2.0
15. International Electrotechnical Commission: IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems (2010), ed2.0
16. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodova, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In: CAV 2009. LNCS 5643, Springer (2009)
17. Kaivola, R., Kohatsu, K.R.: Proof engineering in the large: Formal verification of Pentium 4 floating-point divider. In: Margaria, T., Melham, T.F. (eds.) CHARME. pp. 196–211. LNCS 2144, Springer (2001)
18. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag (1991)
19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conference, DAC'01. pp. 530–535 (2001)
20. de Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340. LNCS 4963 (2008)
21. NoBug: Homepage. `http://nobug.ro`, cited March 2011
22. Pentium FDIV bug. `http://en.wikipedia.org/wiki/Pentium_FDIV_bug`
23. SATabs: Homepage. `http://www.cprover.org/satabs/`, cited July 2011
24. Storey, N.: Safety Critical Computer Systems. Addison-Wesley (1996)

A Tool for the Certification of Sequential Function Chart based System Specifications

Jan Olaf Blech

fortiss GmbH

Abstract. We describe a tool framework for certifying properties of sequential function chart (SFC) based system specifications: CERTPLC. CERTPLC handles programmable logic controller (PLC) descriptions provided in the SFC language of the IEC 61131–3 standard. It provides routines to certify properties of systems by delivering an independently checkable formal system description and proof (called certificate) for the desired properties. We focus on properties that can be described as inductive invariants. System descriptions and certificates are generated and handled using the COQ proof assistant. Our tool framework is used to provide supporting evidence for the safety of embedded systems in the industrial automation domain to third-party authorities. In this paper we focus on the tool's architecture, requirements and implementation aspects.

1 Introduction

Discovering and validating properties of safety critical embedded systems has been a research topic during the last decades. *Automatic verification tools* based on model checking and static analysis techniques are used in various software and hardware development projects. Automatic verification tools are successfully applied to increase confidence in the system design. However, even the verdicts about systems provided by automatic verification tools may be erroneous, since automatic verification tools are likely to contain errors themselves: they use sophisticated algorithms, resulting in complicated implementations. Due to this high level of complexity of their algorithms and the underlying theory, they are hardly ever considered as trustable by certification authorities.

In contrast to general purpose *higher-order theorem provers*, an automatic verification tool possesses a high degree of automation, but it does not achieve the same level of trustability and is usually specialized towards a problem-specific domain. Higher-order theorem provers, like COQ [13], are based on a few deduction rules and come with very small, simple, and trusted proof checkers which are based on type checking algorithms and provide a high level of confidence.

For this reason we provide a verification / certification environment based on higher-order theorem provers. It may be used to re-check properties that have been discovered by automatic verification tools or stated by humans in the first place. If such a check is run successfully in the higher-order theorem prover one lifts these properties to the high level of confidence provided by the higher-order theorem prover.

Based on our ideas on certification of properties for a modeling language [8] and our work on a certificate generating compiler [5] we present a tool framework CERTPLC which emits certificates and allows reasoning about properties of models for programmable logic controller (PLC) provided in the sequential function chart (SFC) language of the IEC 61131–3 standard [17]. Our work comprises a generation mechanism for COQ representations of our models – a kind of compiler that emits COQ readable files for given models. In addition to this, it comprises other related proof generation mechanisms and a framework for supporting proofs that properties of these models do hold. Our COQ certificates – system description, properties and their proofs – are based on an explicit semantics definition of the SFC language, thereby ensuring that correctness conditions hold for the system described in the certificate.

The COQ environment has been accepted by French governmental authorities in a certification to the highest level of assurance of the Common Criteria for Security [12].

In this paper we focus on the following aspects of the CERTPLC tool framework:

– tool architecture,

– proof generation and the construction of certificates,

– and additional implementation issues.

Furthermore, we give an overview on usage scenarios, the formalized SFC semantics and present and discuss general characteristics of the methodology. A long version of this paper is available as a report [4]. In the current state of implementation the tool framework is applicable for standard PLC described using SFC. An exemplary usage with another language: function block diagrams (FBD) is also described to illustrate the flexibility of the described framework. The support of other languages and a detailed investigation of case studies are not subjects of this paper.

Our certification framework is mostly characterized by:

– The usage of an explicit semantics for properties and systems. This is human readable, an important feature to convince certification authorities.

– The focus on the PLC domain and the integration in an existing tool.

– A high degree of automation – compared to other work using higher-order theorem provers, that still allows human interaction.

– The integration into an existing tool for graphically designing PLC: EasyLab [2].

The high expressiveness of our semantics framework is largely facilitated by the usage of a higher-order theorem prover.

1.1 Certification

In the context of this paper we define

- *certification* as the process of establishing a certificate.
- *automatic certification* is the process of establishing a certificate automatically.
- In our work *certificates* comprise a formal description of a system, a formal description of a desired property and a proof description (a proof script or a proof term) that this property does hold.
- *certificate checking* is the process of checking that the property does indeed hold for the formal system description in the certificate. This checking is done by using the proof description in the certificate.

1.2 The Trusted Computing Base in Certification

Apart from components like operating system and hardware, in our certification approach, the trusted computing base (TCB) comprises the certificate checker (the core of the COQ theorem prover) and the program that generates formal PLC descriptions for COQ automatically. The check that these descriptions indeed represent the original PLC can be done manually. One goal for the generation is human readability to make such a check feasible at least for experienced users. Not part of the TCB are the proof description and its generation mechanism. The proof description only provides hints to the certificate checker. In case of faulty proof descriptions a valid property might not be accepted by a certificate checker. It can never occur that a faulty property is accepted even if wrong proof descriptions are used. Thus, our approach is sound, but not necessarily complete.

1.3 Related Work

Notable milestones on frameworks to certify properties of systems comprise proof carrying code [16]. Proofs for program-specific properties are generated during the compilation of these programs. These are used to certify that these properties do indeed hold for the generated code. Thus, users can execute the certified code and have, e.g., some safety guarantees. At least two problems have been identified:

1. Properties have to be formalized with respect to some kind of semantics. This is sometimes just implicitly defined.
2. Proof checkers can grow to a large size. Nevertheless, they have to be trusted.

The problem of trustable proof checkers is addressed in foundational proof carrying code [1, 22]. Here the trusted computing base is reduced by using relatively small proof checkers. The problem of providing a proof carrying code approach with respect to a mathematically founded semantics is addressed in [20]. In previous work we have also addressed the problem of establishing a formal semantics for related scenarios [5, 8].

Formal treatment of PLC and the IEC 61131–3 standard has been discussed by a larger number of authors before. Formalization work on the semantics of the Sequential Function Charts is given in [10, 11]. This work was a starting point for our formalization of SFC semantics.

The paper [3] considers the SFC language, too. Untimed SFC models are transformed into the input language of the Cadence SMV tool. Timed SFC models are transformed into timed automata. These can be analyzed by the Uppaal tool.

Another language of the IEC 61131–3 standard used for specifying PLC are function block diagrams (FBD). Work in the formal treatment of FBD can be found in [23]. The FBD programs are checked using a model-checking approach. A COQ formalization of instruction lists (IL) – also part of the IEC 61131–3 standard – is presented in [18].

The approach presented in [21] regards a translation from the IL language to an intermediate representation (SystemC). A SAT instance is generated out of this representation. The correctness of an implementation is guaranteed by equivalence checking with the specification model.

1.4 Overview

We present the IEC 61131–3 standard, including the SFC language and its semantics as formalized in COQ in Section 2. The tool environment in which our CERTPLC tool framework is supposed to be used and an overview about the tool's architecture is described in Section 3. The CERTPLC ingredients and their interactions are described in some detail in Section 4. Typical proofs that can either be generated or hand-written by using our semantics are discussed in Section 5. Finally, an implementation overview and a short evaluation is given in Section 6. A conclusion is featured in Section 7.

2 IEC 61131–3, SFC, Semantics and Certification

In this section we sketch the semantics of sequential function charts (SFC). The description in this section is based on our earlier work [6] which is influenced by the descriptions in [10, 11]. Furthermore, we present some work on the integration of function block diagrams (FBD) into our tool framework.

2.1 The SFC Language

Our tool framework works with PLC described in the SFC language. The SFC language is a graphical language for modeling PLC. It is part of the IEC 61131–3 standard and frequently used together with other languages of this standard. In such a case, SFC are used to describe the overall control flow structure of a system. The standard is mainly used in the development of embedded systems in the industrial automation domain.

The standard leaves a few semantical aspects open to the implementation of the PLC modeling and code generation tool. In cases where the semantics is not well defined by the standard we have adapted our tool to be compatible with the EasyLab [2] tool.

Syntax Syntactically we represent an SFC as a tuple $(S, S_0, T, A, F, V, \mathit{Val}_V)$. It comprises a set of steps S and a set of transitions T between them. A step is a system location which may either be active or inactive in an actual system state, it can be associated with SFC action blocks from a set A. These perform sets of operations and can be regarded as functors that update functions representing memory. The mapping of steps to sets of action blocks is done by the function F. Memory is represented by a function from a set of variables V to a set of their possible values Val_V. $S_0 \subseteq S$ is the set of initially active steps.

A transition is a tuple (S_{in}, g, S_{out}). It features a set of states that have to be enabled $S_{in} \subseteq S$ in order to take the transition. It features a guard g that has to be evaluated to true for the given system state. g is a function from system memory to a truth value – in COQ we formalize this as a function to the *Prop* datatype. A transition may have multiple successor steps $S_{out} \subseteq S$. The types Val_V that are formalized in our SFC language comprise different integer types and boolean values.

Figure 1 shows an example SFC structure realizing a loop with a conditional branch. The execution starts with an initialization step *Init*. After it has been processed control may pass to either *Step2* or to a step *Return*. Once *Step2* has been processed control is passed to *Init* again.

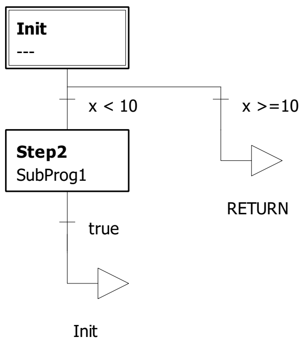

Fig. 1: A loop in the SFC language

Please note, that in addition to loops and branches SFC allow for the formalization of parallel processing and synchronization of control. This is due to the multiple successor and predecessor steps in a transition.

Semantics Semantically the execution of an SFC encounters states, which are (m, a, s) tuples. They are characterized by a memory state m, the function from variables to their values, a set of active action blocks a that need to be processed and a set of active steps s.

The semantics is defined by a state transition system which comprises two kinds of rules:

1. A rule for processing of an action block from the set of active action blocks a. This corresponds to updating the memory state and removing the processed action block from a.
2. A rule for performing a state transition. The effect on the system state is that some steps are deactivated, others are activated. Each transition needs a guard that can be evaluated to true and a set of active steps. Furthermore, we require that all pending action blocks of a step that is to be deactivated have been executed.

It is customary to specify the timing behavior of a step by time slices: a (maximal) execution time associated with it. In our work, this is realized using special variables that represent time.

2.2 The FBD Language

Function block diagrams are a language from the IEC 61131–3 standard used to model the behavior of action blocks in SFC. Other languages that may be used for this purpose comprise instruction lists (IL) and ladder diagrams (LD).

FBD comprise two basic kinds of elements: function blocks and connections between them. Each function block represents an instruction. There are special instructions for reading and writing global variables. Edges between function blocks are used to model dataflow. Thus, FBD are used to describe functions.

Apart from the basic functionality, FBD may contain cycles in their dataflow description. Semantically such a cycle must feature a delay element. Variable values associated with such an FBD are computed in an iterative process.

In the case of cyclic dependencies an FBD has to be associated with a time slice, a maximal time – number of iterations – for the execution of the FBD. Thus, on an abstract level, FBD may still be regarded as functions and as SFC action blocks.

We have formalized an FBD syntax and semantics framework in COQ that follows the description above. Most parts of this, however, are only to be used manually by users who manually change system descriptions and corresponding proofs.

3 The Tool Setting

In this section we describe our CERTPLC tool's architecture and usage scenarios. Figure 2 shows the CERTPLC ingredients and their interconnections. In an invocation of

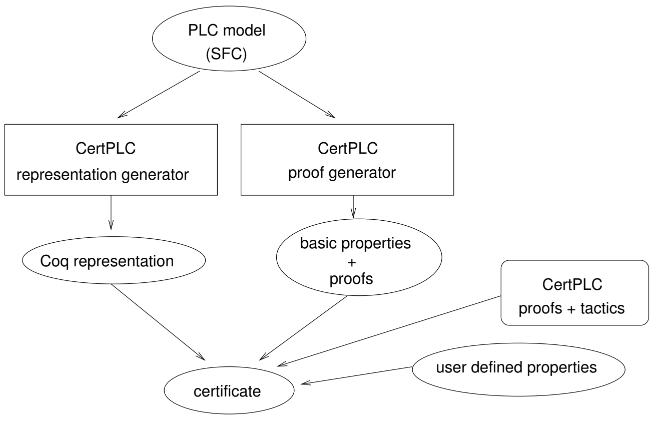

Fig. 2: CERTPLC overview

the tool framework an SFC model is given to a

- **representation generator** which generates a COQ representation out of it. This is included in one or several files containing the model specific parts of the semantics of the SFC model. The COQ representation is human readable and can be validated against the original graphical SFC specification by experienced users.

The same SFC model is given to a

- **proof generator** which generates COQ proof scripts that contain lemmas and their proofs for some basic properties that state important facts needed for machine handling of the proofs of more advanced properties.

In order to achieve a certificate one needs a property that the certificate shall ensure. One needs to formalize this desired property. The property is proved in COQ by using either a provided tactic or a hand written proof script. Our provided tactics use the generated properties and their proofs – provided by the proof generator – and a collection of

– **proofs and tactics**, a kind of library. It contains additional preproved facts and tactics which may be used to automatically prove a class of properties.

System description, used lemmas and their proofs, and the property and its proof form a certificate.

Furthermore, our tool framework comprises a COQ library that can be used by generated and non-generated COQ files. It allows storage of often used definitions in addition to the elements described so far. We have formalized some behavioral definitions of PLC blocks which are typically modeled in other languages than SFC.

Usage Scenario

CERTPLC is developed to support the following standard usage scenario:

– A PLC is developed using the following work-flow:
  1. Establishing requirements,
  2. and derive some early formal specification.
  3. Based on this specification the overall structure – e.g., the control flow – is specified using the SFC language. More fine-grained behavioral aspects are textually specified, e.g., by annotating the SFC structure.
  4. Taking the requirements and this specification, developers potentially using the help of automatic verification tools derive and specify consistency conditions and properties that must hold. Some consistency conditions may directly correspond to a subset of the requirements.
– Regarding 3) the SFC structure is modeled in the graphical EasyLab tool or imported into EasyLab.
– Regarding 4) properties and SFC action blocks are specified using the COQ syntax by trained developers. It is not required to do any proofs in COQ for this.
– CERTPLC generates representations for the PLC specification. Together with the properties a certificate is established automatically or with user interaction: the choice of tactics and in some-cases hand-written proof script code.
– The PLC development is further refined and fine grained parts may be implemented using other languages from the IEC 61131–3 standard.
– Certificates may be either regenerated – if possible – or manually adapted – in case of unsupported language elements that may occur during the refinement – to cope with possible changes.

The certificate can be distributed and analyzed independently by third parties. One overall goal is to convince certification authorities and potential customers of the correctness of PLC with the help of certificates. Since the certificates are independent of the original development and its tools some confidential data (e.g., the certificate generation mechanism and the analysis algorithms used to discover properties of the system) does not have to be revealed during the process of convincing customers or certification authorities.

The described usage scenario can be adapted. It is, e.g., possible to integrate hand written specifications and proofs.

4 The CertPLC Tool Environment and Coq

In this section we describe COQ specific parts of the CERTPLC tool. We present some static COQ code that is generic to our framework. Furthermore, we present some PLC specific example COQ code – definitions and proofs – to demonstrate aspects of its generation.

Taking the semantics sketch of SFC in Section 2 the semantic representation of the SFC structure is encoded in COQ as a transition system. For each given SFC SFC we generate a COQ representation. It specifies a set of reachable states and a transition relation.

4.1 Realization Using Generic and Generated Files

In order to certify properties of PLC we need files that contain semantics of systems, interesting properties and proofs of these properties. Some of these files are generic, i.e., they can be used for a large class of PLC, properties, and proofs. CERTPLC provides a library of static files that contain generic aspects. Other files are highly specific to distinct PLC. For each PLC CERTPLC generates files that are just needed for this particular PLC, properties formulated on it, and proofs that can be conducted on it.

In particular the following aspects are generic, thus, stored in static files:

- Generic definitions and templates for SFC:
  - Datatype definitions and derived properties of these datatypes.
  - Definitions for building blocks: SFC action blocks, FBD blocks, and common combinations of these blocks.
  - Generic semantics framework comprising an instantiable state transition relation and a generic definition for a set of reachable states.
- CERTPLC is designed to support tactics for solving certain proof aspects. In particular we distinguish:
  - Tactics that contain an overall proof structure, deal with certain system structures and property structures.
  - Tactics that solve arithmetic constraints.

The following aspects are individual for each PLC, thus, they are generated:

- A state transition like representation of SFC formalized using generic SFC definitions and a concrete definition of reachable states instantiating a generic SFC definition.
- Lemmas containing system-specific facts on the PLC and their proofs.

Furthermore, the properties that shall hold are of course specific to each PLC. Their verification is done by either using a tactic that assembles the generic and non-generic parts of the proof or by some hand-written proof script adaptations.

4.2 Generic / Static Parts of the Coq Infrastructure

Here we describe generic parts of the COQ parts in our CERTPLC tool framework. These are realized as static COQ files and can used by the dynamically created files.

Datatypes Datatypes which we have formalized for SFC comprise integers of different length (8,16,32 bit, unbounded) and bools. In COQ they are stored using the datatype *nat* of natural numbers plus a flag that tags them as being members of an integer type. Operators working on these integers perform operations compliant with the type. An easy integration of other bounded integer formalizations (as used e.g., in [15]) is also possible.

Other datatypes like floating point are seldomly used in PLC applications. They are not yet supported, although they could be integrated relatively easy: The basic semantics definitions in our framework are able to support a much richer type system, even dependent datatypes.

Building Blocks Building blocks define common elements for the construction of PLC. Two levels of building blocks can be distinguished:

- Function blocks that are intended to become part of FBD.
- Predefined action blocks. These may be, but do not have to modeled using FBD.

As mentioned in Section 3 we have formalized some of these blocks. Further formalization of blocks should be done together with new case studies since different application domains have different sets of FBD and SFC elements. FBD elements that are highly specific to a single application or an application domain are highly common in PLC. For FBD we have experienced even vendor specific elements for the basic arithmetic operations.

Generic Semantics Framework The COQ realization of the SFC syntax follows the description presented in Section 2. For compatibility with the EasyLab tool and to ease generation we distinguish between steps and step identifiers in our COQ files, thereby introducing some level of indirection. Most importantly, our semantics framework comprises a template for a state transition relation of PLC systems and a template for defining the set of reachable states. In order to realize this, we first define generic instantiable predicates that formalize a state transition relation. We provide a predicate *executeAction* defined in Figure 3 to give a look and feel. It formalizes the effect of the execution of an action block: The predicate takes two states (sometimes called configurations c and c') and returns a value of type *Prop*. We require four conditions to hold in order to take a state transition:

1. An action block a needs to be active.
2. The memory mapping after the transition is the application of a to the previous memory mapping. This is the updating of the memory by executing the action block.
3. The action block a is removed from the set of active action blocks during the transition.

```
Definition executeAction:
   fun c c' =>
      let '(m,activeA,activeS) := c in
      let '(m',activeA',activeS') := c' in
         (exists a, In a activeA /\ m' = a m /\
          activeA' = remove Action_eq_dec a activeA) /\
          activeS = activeS'.
```

Fig. 3: The *executeAction* predicate

4. The rest of the state does not change.

Another predicate *stepTransition* formalizes the effect of a transition from a set of SFC steps to another. Here we require the following items:

1. The validity of the transition (guard expression).
2. The memory state may not change.
3. The activation of steps is conform to the semantics.
4. The activation and requirements of action blocks is semantics conform.

Using these predicates we define inductively the set of reachable states as a predicate. It depends on an initial state (comprising a list of initially active steps), and a transition relation. It is defined following the description in Section 2.

Structural Tactics CERTPLC supports structural tactics that perform the most basic operations for proofs of properties. They work with semantics definitions based on our generic semantics framework. Depending on the property such a tactic is selected by the user and applied as the first step in order to prove the desired property. Different tactics have to be selected by the user: Selection depends on whether the property is some kind of inductive invariant – the default case mostly addressed in this paper – or another class of properties. We have identified several other classes that are relevant for different application domains. Such a tactic is applied as the first step in order to prove the desired property. These tactics already perform most operations concerning the system structure. Especially for the non-standard cases, tactics applications may leave several subgoals open. These may be handled with more specialized tactics tailored for the corresponding characteristics of these proof-goals.

Arithmetic tactics Arithmetics tactics solve subgoals that appear at later stages in the proof. They may be called by structural tactics or work on open subgoals that are left open by these tactics. They comprise classical decision procedures like (e.g., Omega [19] – its implementation in COQ).

Up till now, we are only using existing tactics designed by others. However, we are also working on tactics that combine arithmetic aspects with other system state dependent information.

4.3 Semantics Definitions as State Transition Systems

As seen in Section 4.2 we only need to instantiate a template in order to create a system definition that captures the semantics of our PLC. We need to provide at least a set of initially active steps, a transition relation, and action block definitions.

For the initial step, we provide an initial memory state, where all values are set to a default value and a single active entry step.

The transition relation is generated by translating the SFC transition conditions into COQ. The generated COQ elements of the transition relation for the SFC depicted in Figure 1 are shown in Figure 4. Three tuples are shown, each one comprises a set of activated source steps, a condition and a set of target steps activated after the transition. It can be seen that the condition maps a variable value mapping – part of the SFC state – to a truth condition – returning the type *Prop*. The types used in this expression are 16-bit integer types.

```
( Init::nil ,
  fun m => ((fun  (x : int16) => x <int16 10 )  (m VAR_x) ),
  Step2::nil )

( Step2::nil ,
  fun m => ((fun  (x : int16) => 1 )  (m VAR_x) ),
  Init::nil )

( Init::nil ,
  fun m => ((fun  (x : int16) => x >=int16 10 )  (m VAR_x) ),
  Return::nil )
```

Fig. 4: Generated transition rules in COQ

Appropriate action blocks are selected by their names. In addition, to this, we generate several abstract datatype definitions for identifying steps with names and identifiers and function blocks and action blocks.

4.4 Automatically Generated Proofs for System-specific Facts

CERTPLC is designed to automatically generate for each system basic properties and proofs. These prove some system-specific facts of the system. These proofs are used automatically by tactics, but can also be used manually to prove additional user defined properties of systems.

One important fact that needs to be proven is that only those action blocks may appear in the set of currently active action blocks that do belong to the actual system definition. Our proof generator generates an individual lemma and its proof for each PLC. Figure 5 shows such a lemma for an SFC that comprises just two possible action blocks: *action_Init* and *action_Step2*. The predicate *SFCreachable_states* is created by instantiating a template definition from the generic semantics framework for a concrete

```
Lemma aux_1:
   forall s, SFCreachable_states s -> (forall a, In a (snd s) ->
         (  a = action_Init \/  a = action_Step2 )).
```

Fig. 5: An automatically generated basic property

PLC. *In* and *snd* (second) are COQ functions to denote membership in a set and select an element of a tuple, respectively. In the case at hand *snd* selects the set of active action blocks from a state. The proof script itself is also generated. It comprises an induction on reachable states of the concrete system. Depending on the number of action blocks in the PLC it can typically comprise several hundred applications of elementary COQ tactics.

The certification of properties is the key feature of CERTPLC. Users write their desired properties in COQ syntax. This does not require as much understanding of the COQ environment as one could think at a first glance. All that is required is writing a logical formula that captures the desired property.

5 Automatic Certification of Invariant Properties

In this section we describe the principles of automatically proving properties correct. Proof scripts encapsulating these principles are generated by the CERTPLC framework components as described in Section 4. We focus on inductive invariants.

5.1 Proof Structure for Inductive Properties

We start with an inductive invariant property I and an SFC description of a PLC SFC. Following the ideas presented in [8] the structure of a proof contained in our certificates is realized by generated proof scripts, generic lemmas and tactics. They establish a proof principle that proves the following goal:

$$\forall\ s\ .\ s \in \mathit{Reachable}_{SFC} \Longrightarrow I(s)$$

The set of reachable states for SFC is denoted $\mathit{Reachable}_{SFC}$. $[\![SFC]\!]$ specifies the state transition relation (cf. Section 4). First we perform an induction using the induction rule of the set of reachable states. This rule is automatically established by COQ when defining inductive sets. After the application the following subgoals are left open:

$$I(s_0) \text{ for initial states } s_0 \qquad\qquad I(s) \wedge (s, s') \in [\![SFC]\!] \Longrightarrow I(s')$$

The first goal can be solved in the standard case by a simple tactic which checks that all conditions derived from I are fulfilled in the initial states.

For the second goal the certificate realizes a proof script which – in order to allow efficient certificate checking – performs most importantly the following operations:

- Splitting of conjunctions in invariants into independently verifiable invariants.

– Splitting of disjunctions in invariants into two independently verifiable subgoals.
– Normalizing arithmetic expressions and expressions that make distinctions on active steps in the SFC.
– Exhaustive case distinctions on possible transitions. Each case distinction corresponds to one transition in the control flow graph of the SFC. A typical case on a transition from a partially specified state s to a partially specified succeeding state s' can have the following form:

  $\forall\, s\, s'$.
  $I(s)$ and case distinction specific conditions on $s \wedge$
  case specific transition conditions that need to be true to go from s to $s' \wedge$
  case distinction specific definition of $s' \quad \Longrightarrow \quad I(s')$

  The case distinction specific parts in such a goal can, e.g., feature arithmetic constraints, which can be split into further cases.
  Some of the cases that occur can have contradictions in the hypothesis. Consider for example an arithmetic constraint for a variable from a precondition of a state contradicting with a condition on a transition. These contradictions result from the fine granularity of our case distinctions. Some effort can be spent to eliminate contradicting cases as soon as possible (cf. [8]) which can speed up the checking process. Unlike in classical model-checking we get the abstraction from (possibly infinite) concrete states to (finite) arcs in the control flow graph almost for free. Thus, in our case distinctions, we do not have to regard every possible state, we rather partition states into classes of states and reason about these classes symbolically.
– The final step comprises the derivation of the fact that the invariant holds after the transition from the transition conditions and the decision of possible arithmetic constraints.

[8] features a completeness result for a class of inductive invariants for a similar problem.

5.2 Proving Non Inductive Invariants

The main focus of CERTPLC is on inductive invariants, However, additionally we have established a collection of preproved lemmas useful for proving the (un-)reachability of certain states. In particular the following cases turned out to be necessary in our case studies:

– State s can only be reached via a transition where a condition e must be enabled, s is not initial, e can never be true in the system, this implies s can not be reached.
– Under system specific preconditions: Given an expression over states e, if e becomes true the succeeding state will always be s. This is one of the few non-inductive properties. However, the proof of this benefits from a proof that e can only become true in an explicitly classified set of states. This can be provided by one of the techniques above.

Additional consistency properties may be certified by hand-written proof scripts. This, however, requires some level of expertise in COQ.

6 Additional Implementation Aspects and Evaluation

Here we describe additional implementation aspects that are not covered in the previous sections and provide a short evaluation.

The COQ representation generator is implemented as an Eclipse plug-in in Java using the IEC 61131–3 meta model of EasyLab and the Eclipse Modeling Framework (EMF) [14]. Representations and lemmas + proofs for basic properties are generated for COQ 8.3. Likewise our libraries for tactics, lemmas and SFC action blocks are formalized using this version. The realization of this representation generator can be regarded as a simple compiler or model to model transformation. A kind of visitor pattern is used to pass through the model representation in EMF format and emit corresponding COQ code. The generation of PLC specific lemmata and their proofs is similar to code generation. A visitor picks all necessary information and generates the lemma text and its proof script. Some storage of intermediate information is needed. The setup is similar to the techniques used in [8] and [9].

Likewise our work builds upon the PLC semantics of EasyLab which we have formally described [6] and realized in COQ. A combination of our SFC semantics with a semantics of the instruction list (IL) language and an associated case study can be found in [7].

7 Conclusion and Future Work

In this paper we have presented the CERTPLC environment for certification of PLC We described the architecture of the tool framework, possible usage scenarios, the technical realization, and parts of the COQ semantics. CERTPLC is aimed at the formal certification of PLC descriptions in the SFC language. Nevertheless, some features of FBD are integrated. Future work shall extend this support and aims at integrating other languages from the IEC 61131–3 standard. At the current state, the implementation of the tool is sufficient to handle SFC comprising standard elements and smaller invariants efficiently. We believe that the generic parts common to most SFC verification work are realized in CERTPLC. The tool framework is designed such that it is easily extendable, e.g., with additional tactics, arithmetic decision procedures and building blocks for SFC and FBD elements. Such additions – which might be used only in certain problem and application domains – are subject to future work.

Acknowledgments

This work has been supported by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208.

References

1. A. W. Appel. Foundational proof-carrying code *Logic in Computer Science*. IEEE Computer Society, 2001. (LICS'01).

2. S. Barner, M. Geisinger, Ch. Buckl, and A. Knoll. EasyLab: Model-based development of software for mechatronic systems. Mechatronic and Embedded Systems and Applications, IEEE/ASME, October 2008.
3. N. Bauer, S. Engell, R. Huuck, B. Lukoschus, and O. Stursberg. Verification of plc programs given as sequential function charts. In *In: Integration of Software Specification Techniques for Applications in Eng.*, pages 517–540, Springer-Verlag, 2004.
4. J. O. Blech. A Tool for the Certification of PLCs based on a Coq Semantics for Sequential Function Charts. http://arxiv.org/abs/1102.3529, 2011.
5. J. O. Blech and B. Grégoire. Certifying Compilers Using Higher Order Theorem Provers as Certificate Checkers. Formal Methods in System Design, Springer-Verlag, 2010.
6. J. O. Blech, A. Hattendorf, J. Huang. An Invariant Preserving Transformation for PLC Models. Model-Based Engineering for Real-Time Embedded Systems Design, IEEE, 2011.
7. J. O. Blech and S. Ould Biha. Verification of PLC Properties Based on Formal Semantics in Coq. Software Engineering and Formal Methods, 2011. (SEFM'11) *accepted*
8. J. O. Blech and M. Périn. Generating Invariant-based Certificates for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS). *accepted*
9. J. O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation validation for system abstractions. In *Runtime Verification* , vol. 4839 of *LNCS*. Springer-Verlag, March 2007. (RV'07)
10. S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. An Abstract Model for Sequential Function Charts. Discrete Event Systems: Analysis and Control, Workshop on Discrete Event Systems, 2000.
11. S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. Verification of Sequential Function Charts using SMV. Parallel and Distributed Processing Techniques and Applications. CSREA Press, June 2000. (PDPTA 2000)
12. B. Chetali and Q. H. Nguyen. Industrial Use of Formal Methods for a High-Level Security Evaluation. *Formal Methods in the Development of Computing Systems*, vol. 5014 of *LNCS*. Springer-Verlag, 2008.
13. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.3*, 2010. http://coq.inria.fr.
14. The Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/.
15. X. Leroy. A formally verified compiler back-end. In *Journal of Automated Reasoning*, Vol.43, No.4, pp.363-446, 2009.
16. G. C. Necula. Proof-carrying code. *Principles of Programming Languages*. ACM Press, 1997. (POPL'97).
17. Programmable controllers - Part 3: Programming languages, IEC 61131-3: 1993, International Electrotechnical Commission, 1993.
18. S. Ould Biha. A formal semantics of PLC programs in Coq. IEEE Computer Software and Applications Conference, 2011.
19. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *ACM/IEEE Conference on Supercomputing*, 1991. (SC'91).
20. R. R. Schneck and G. C. Necula. A Gradual Approach to a More Trustworthy, Yet Scalable, Proof-Carrying Code. *Conference on Automated Deduction*, vol. 2392 of *LNCS*, Springer-Verlag, 2002. (CADE'02).
21. A. Sülflow and R. Drechsler. Verification of plc programs using formal proof techniques. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, Budapest, 2008.
22. D. Wu, A. W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. *ACM Conference on Principles and Practice of Declarative Programming*. ACM Press, 2003. (PPDP'03).
23. J. Yoo, S. Cha, and E. Jee. Verification of plc programs written in fbd with vis. In *Nuclear Engineering and Technology*, Vol.41, No.1, pp.79-90, February 2009.

Author Index