Adaptable Value-Set Analysis for Low-Level Code

Jörg Brauer[1], René Rydhof Hansen[2], Stefan Kowalewski[3], Kim G. Larsen[2], and Mads Chr. Olesen[2]

1 **Verified Systems International GmbH**
Am Fallturm 1, 28359 Bremen, Germany

2 **Department of Computer Science, Aalborg University**
Science Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark

3 **Embedded Software Laboratory, RWTH Aachen University**
Ahornstr. 55, 52074 Aachen, Germany

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.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning About Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases Abstract interpretation, SAT solving, embedded systems

Digital Object Identifier 10.4230/OASIcs.SSV.2011.32

1 Introduction

Model checking and abstract interpretation [15] 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 [11], 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 [15] is to abstract away from the detailed nature of states, and rather represent sets of concrete states using geometric concepts such as affine [20] or polyhedral spaces [16]. A program analyzer then operates over classes of states that are related in some sense — for example, sets of states that are described 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 system. However, sometimes much detail is lost when working with abstract classes so that the technique cannot infer

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

Figure 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.

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 [26, 32], 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, 21, 22, 32]. 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 [12], such encodings can straightforwardly be derived.

1.1 Value-Set Analysis using SAT

This paper studies and extends 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 on the involved registers. 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 obtain 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. These encodings can then, e.g., be composed to represent the semantics of basic blocks. Although supporting the entire instruction set 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 the sheer number of different instructions 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 [18], 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 range analysis based on binary search [13]. Let φ denote a Boolean formula that characterizes a bit-vector $\boldsymbol{r}$. From an encoding ψ of an instruction or basic block, φ can be obtained by projecting ψ onto $\boldsymbol{r}$, e.g., using incremental SAT solving [8] or BDDs [23].

2.1 Range Analysis using Incremental SAT

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 bound to the range $0 \leq \langle\langle \boldsymbol{r} \rangle\rangle \leq 2^w - 1$, and so is $\langle\langle \boldsymbol{r}_\ell^1 \rangle\rangle$. Since $\langle\langle \boldsymbol{r}_\ell^1 \rangle\rangle$ is uniquely determined, 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:

$$\begin{array}{lcl} \mu_\ell & = & 0 \leq \langle\langle \boldsymbol{r}_\ell^1 \rangle\rangle \leq 2^{w-1} - 1 \\ \mu_u & = & 2^{w-1} \leq \langle\langle \boldsymbol{r}_\ell^1 \rangle\rangle \leq 2^w - 1 \end{array}$$

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 tactic 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

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

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:

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

2.2 Value-Set Analysis using Incremental SAT

The key idea of VSA is then to repeatedly apply this technique so as to alternatingly remove ranges from and add ranges to 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* within $\mathcal{V}^1$ and removes this range from $\mathcal{V}^1$. This gives an under-approximation $\mathcal{V}^2$ 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 returns the value-set $\mathcal{V}^k$.

3 Worked Example

Our approach is to first translate each instruction in a program from a hardware-specific representation into an intermediate language. Liveness analysis [34] 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; this is due to side-effects on the status word, which includes flags such as the carry oder negative flags. Many instructions have side-effects, yet, few of them actually influence the behavior of the program. The elimination of dead operations is followed by a conversion of each block into static single assignment (SSA) form [17]. 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 [31].

Table 1 Operand-size identifiers used in the IR.

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`

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 operand-size identifiers given in Tab. 1. In our 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 `ANDI R0 #15` (logical-and with immediate) 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, p. 20], which we translate the following Boolean formula:

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

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` in location `0x43`):

```
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 [28] and in order to eliminate redundant assignments, which do not have any effect on the program execution. This technique typically simplifies the intermediate program representations — 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 benchmarks. It is thus meaningful with respect to tractability to decouple the explicit effects of an instruction from its side-effects. Of course, the effectiveness of liveness analysis depends on the target architecture.

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 [12]. 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:

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

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 [24] are equally applicable.

3.3.1 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 $\boldsymbol{o}$, denoted $\pi_{\mathbf{0}}(\varphi)$. Then, $\pi_{\boldsymbol{o}}(\varphi)$ describes a Boolean relationship between $\boldsymbol{r0}'$ and the carry flag $\boldsymbol{r95}'[0]$.

3.3.2 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`. 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 $0 \leq \langle\!\langle \boldsymbol{o} \rangle\!\rangle \leq 2^9 - 1$. VSA then yields the following value-sets:

$$\langle\!\langle \boldsymbol{o} \rangle\!\rangle \quad \in \quad \left\{ \begin{array}{l} 220, 222, \ldots, 252, 254, \\ 256, 258, \ldots, 268, 270 \end{array} \right\}$$

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 are interpreted as follows:

- 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.
- The value-set $\langle\!\langle \boldsymbol{r0}' \rangle\!\rangle \in \{256, 258, \ldots, 268, 270\}$ is reduced by 256 to eliminate 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 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 introduced in Sect. 3.3 can likewise be applied to derive relations between different bit-vectors that represent different registers. This tactic gives a weakly relational value-set representation. Suppose we apply 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 $\boldsymbol{r0}'$ on output can take for inputs such that either

$$0 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 127$$

or

$$128 \leq \langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 255$$

holds. If VSA yields a value such that the most significant bit of $\boldsymbol{o}'$ is set, then $\langle \boldsymbol{o}'[0], \ldots, \boldsymbol{o}'[7] \rangle$ determines a value that is reachable if $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \geq 128$. Likewise, if $\boldsymbol{o}'[0]$ is cleared, then

$\langle \boldsymbol{o}'[0], \ldots, \boldsymbol{o}'[7] \rangle$ determines values reachable if $\langle\!\langle \boldsymbol{r0} \rangle\!\rangle \leq 127$. Clearly, 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 reachable output values according to the inputs:

- $0 \leq 63$,
- $64 \leq 127$,
- $128 \leq 191$, and
- $192 \leq 255$.

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 [25]. 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 [33].
- How does analyzing extended bit-vectors affect the overall performance compared to the SAT-based analysis discussed in [31]. 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 [31, 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 [31], 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.

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 [18]

Table 2 Experimental results for SAT-based VSA using extended bit-vectors.

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

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 [15], 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 [16, 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 [19] 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, 21, 22, 26, 29, 30, 32]. In their seminal work, Reps et al. [32] 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 [22] and affine relations [5]. Both approaches replace the theorem prover from [32] 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. [32], it was observed by Regehr et al. [29, 30] 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 [24, 27], 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. [13] 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 [31] 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. [30]), we have decided to synthesize transfer functions (or abstractions, respectively) in our tool [mc]square [33] 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 [22, Sect. 4] or [14, 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 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 *CAV*, 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*, volume 6806 of *LNCS*, pages 165–170. Springer, 2011.
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*, volume 6806 of *LNCS*, pages 463–469. Springer, 2011.

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. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194. Springer, 2001.

12 E.W. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

13 M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.

14 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.

15 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.

16 P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.

17 R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, pages 451–590, 1991.

18 A. E. Dalsgaard, M. Chr. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *WCET*, pages 113–123, 2010.

19 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.

20 M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

21 A. King and H. Søndergaard. Inferring congruence equations using SAT. In *CAV*, volume 5123 of *LNCS*, pages 281–293. Springer, 2008.

22 A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.

23 S. K. Lahiri, R. E. Bryant, and B. Cook. A Symbolic Approach to Predicate Abstraction. In *CAV*, volume 2725 of *LNCS*, pages 141–153. Springer, 2003.

24 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.

25 D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. `http://www.sat4j.org/`.

26 D. Monniaux. Automatic Modular Abstractions for Linear Constraints. In *POPL*, pages 140–151. ACM Press, 2009.

27 D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.

28 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.
29 J. Regehr and U. Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES*, pages 34–43. ACM, 2006.
30 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.
31 T. Reinbacher and J Brauer. Precise Control Flow Recovery Using Boolean Logic. In *EMSOFT*, pages 117–126. ACM Press, 2011.
32 T.W. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
33 B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4):1–27, 2010.
34 B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on Binary Rewriting (WBT 2001)*, 2001.