SAT-Based Abstraction Refinement for Programmable Logic Controllers

Sebastian Biallas, Jörg Brauer, Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Email: {lastname}@embedded.rwth-aachen.de

***Abstract*—This paper studies the application of counterexample-guided abstraction refinement to programs written in Instruction List. More importantly, it presents an approach for automatic abstraction refinement based on SAT solving. This technique is based on an encoding of the semantics of Instruction List in propositional Boolean logic. Since elegant ideas and careful engineering have advanced SAT solvers to the state they can rapidly decide satisfiability of structured problems that involve thousands of variables, this approach scales well in practice. The true force of this method, however, is that a single description of the semantics of a program can be used to perform abstraction refinement in a number of abstract domains, including but not limited to intervals and bit sets, thereby decoupling the refinement from the chosen abstraction.**

I. Introduction

Programmable logic controllers (PLCs) are frequently used to control safety-critical systems. Since failure of such systems may have disastrous effects, the application of formal methods to the software of such systems is highly desirable, if not even recommended [1]. Two particular techniques that allow to prove the absence of runtime errors are model checking [2] and abstract interpretation [3]. In model checking, the behavior of a system is formally specified with a model, and all paths through the program are then exhaustively checked against its requirements. The detailed nature of the requirements entails that the program is simulated in a fine-grained way, sometimes down to the level of individual bits. Because of the complexity of this reasoning, there has been much interest in operating over collections of states — which are equivalent in some sense — rather than individual states. This approach ties model checking to abstract interpretation [3], the key idea of which being to simulate the execution of each concrete operation in a program using an abstract counterpart over a collection of states. The notion of abstraction, however, entails an inevitable loss of precision. If the specification of the system is satisfied on an abstract semantics of the program, then it is also satisfied by the concrete program. The converse, however, does not hold. If the abstract system violates the specification, this may be due to abstraction, which manifests itself in spurious counterexamples.

This observation has led to the development of a technique called counterexample-guided abstraction refinement (CEGAR) [4], [5], [6]. The key idea of this approach is to automatically refine the abstract semantics of a program whenever a spurious counterexample is encountered. The method terminates if either the refined abstract model satisfies the specification or a real counterexample is found. Implementing a refinement loop which suppresses certain undesired behavior, however, is a challenging task. This is because abstract states of a program are classically represented using geometric concepts such as intervals [3], polyhedral spaces [7], or affine spaces [8]. This conceptual difference between the fined-grained nature of states — which consist of collections of bits — and high-level geometric abstractions presents a semantic gap that has to be bridged. Automatic abstraction, and accordingly automatic refinement, presents a technique to overcome this gap.

A. Overview

Our model checking framework focusses on software written for PLCs which operate in the cyclic scanning mode. In general, the cyclic scanning mode consists of three atomic steps: (1) reading inputs which are usually connected to sensors, (2) performing a computation and (3) writing outputs which are usually connected to actuators. These steps are executed at a high frequency and the PLC can be considered as a black box which sets output variables depending on input values (and internal variables that last over multiple cycles, which we refer to as *global* variables). As all steps are atomic, the program might assume invalid states before the end of the cycle without violating its specification — the output connectors are not changed until the program has finished its computation. On the one hand, this behavior has to be taken into account when verifying the program, which poses a particular challenge. On the other hand, knowledge about the structure of the system also allows to design domain-specific verification techniques which turn out to be more efficient than general-purpose approaches. Recent work [9], [10] has shown this for programs written in Instruction List (IL) [11].

B. Approach

To allow for the verification of PLC programs, we use a direct model checking approach according to [9]. Since PLCs programs usually have a huge number of inputs, they are especially prone to state explosion, which we cope with using an abstract simulation and refinement technique described in [10]. The simulation starts with the most general abstract values for all inputs and successively refines them at key points in the program, taking account of the cyclic execution mode, whenever it is required to do so. These points are guarded with constraints at the following program locations:

- All conditional instructions are guarded such that all branches are taken deterministically to maintain the atomic execution of a cycle .
- At the end of the cycle, some global variables are constrained to suppress spurious counterexamples.
- Variables used in atomic propositions are constrained to decide the truth value of the formula.

To find suitable refinements, a tailored constraint solver was used in [10] to derive constraints on the inputs that suppress the undesired behavior. This constraint solver, however, cannot appropriately handle constrains involving more than one variable. Further, carefully engineering a solver that performs optimal transformations to derive refined input constraints is a time-consuming and error-prone task. This is even more so since the hand-written solver is domain-dependent, and thus needs to be adopted for each abstract domain.

C. Contribution

To efficiently handle more complex constraints and automate the process of refinement, this paper introduces a SAT solving technique which is used to refine the ranges of input variables. Therefore, we interpret each variable $\boldsymbol{x}$ of the program as a vector of propositional (Boolean) variables $\langle x_{n-1}, \ldots, x_0 \rangle$ representing the bits. The most significant bits (MSB) of $\boldsymbol{x}$ is labelled x_{n-1} and we have $\boldsymbol{x} = \sum_{i=0}^{n-1} 2^i \, x_i$.

We then translate the semantics of a given IL program into propositional Boolean logic, a technique that is colloquially referred to as bit-blasting [12]. The resulting Boolean formula describes the relations between input and output variables. A SAT solver is then called iteratively, performing a kind of dichotomic search, to infer upper and lower bounds for variables subject to some constraint. This allows us to automatically derive abstraction refinements for constraints involving multiple variables and complex arithmetic or logical expressions.

D. Structure

The remainder of this paper is structured as follows. First, Sect. II demonstrates the application our approach to a small PLC example program. In Sect. III the individual steps to apply the SAT solving process for range analysis are detailed. The paper ends with an extensive list of related work in Sec. IV and a concluding discussion in Sect. V.

II. Worked Example

We motivate our approach with the example program shown in Fig. 1. This program has two inputs $\boldsymbol{x}$, $\boldsymbol{y}$ and one output $\boldsymbol{z}$, all of type `BYTE` (ranging from 0 to 255). In each cycle $\boldsymbol{z}$ is set to $\min(\boldsymbol{x} + \boldsymbol{y}, 3)$: The sum of $\boldsymbol{x}$ and $\boldsymbol{y}$ is calculated in lines 10 and 11. If it is less than 3 (line 12), the program conditionally jumps to `label`, where the sum of $\boldsymbol{x}$ and $\boldsymbol{y}$ is stored in $\boldsymbol{z}$. Otherwise, 3 is loaded in the accumulator and stored in $\boldsymbol{z}$ (line 13 and 14).

One way to enable the verification of such a PLC program is to apply model checking to the state space of the program. Here, a state refers to the configuration (a tuple representing the values of all variables, inputs and outputs) of the PLCs

```
1    PROGRAM PLC_PRG
2    VAR_INPUT
3           x,
4           y:  BYTE;
5    END_VAR
6    VAR_OUTPUT
7           z: BYTE;
8    END_VAR
9           LD     x
10          ADD    y
11          LT     3
12          JMPC   label
13          LD     3
14          ST     z
15          RET
16  label:  LD     x
17          ADD    y
18          ST     z
19          RET
20  END_PROGRAM
```

Fig. 1. Example program

after the execution of a cycle. As an example, a state of the program in Fig. 1 could be $(\boldsymbol{x} = 0, \boldsymbol{y} = 2, \boldsymbol{z} = 2)$. We can generate a successor of a state by assigning new input values and simulating the PLCs cycle, thus finding new outputs. If we repeat this step for all states and all possible input values, we generate a Kripke structure that contains all reachable states of the program and their connecting transitions. Using a model checker, we now can prove certain properties of this structure using logical formulae. We could prove, for instance, that no invalid state is reachable or that a safety state is reachable from all states.

The limiting step here is the generation of the state space. Even for the example program, each state has 65536 successors, due to the two input variables, which can take values ranging from $0..255$. Each additional input would exponentially increase this number, which is the well-known state explosion problem [13] of model-checking.

To alleviate this problem, different abstraction techniques have been proposed. The keys idea of those abstractions is to combine a set of states that can be treated similarly into a macro state. Consider, for instance, the variable $\boldsymbol{x}$ of the program. Whenever this variable is bound to the interval $[3, 255]$, the conditional branch in line 13 is not taken, always resulting in a state where $\boldsymbol{z}$ is 3. Since addition is symmetric, this also applies to $\boldsymbol{y}$. Thus, we only have to generate successors where $\boldsymbol{x}$ and $\boldsymbol{y}$ assume the intervals $[0, 0]$, $[1, 1]$, $[2, 2]$ and $[3, 255]$, drastically reducing the number of successors.

In the next sections we will investigate how to derive suitable intervals automatically.

A. Refinement

In previous work [10], we describe a refinement technique for abstract values which is based on solving constraints. Our simulation starts with the most general abstract value (intervals ranging over the complete domain), which then are subsequently refined using guards at certain points in the program.

To find suitable refinements, the program is transferred into a *static single assignment* (SSA) form [14]. A constraint on an intermediate value can then be symbolically rewritten into a constraint on an input variable. This input variable is refined into an abstract value representing less concrete values, removing the problematic intermediate value at the guard after restarting the cycle with new values.

In the example program, the conditional jump in line 13 demands a concrete value in the accumulator (the PLC execution mode forbids non-deterministic execution during the cycle) and would therefore be guarded by a constraint. That is, the expression $(\boldsymbol{x} + \boldsymbol{y} > 3)$, which we can extract out of the SSA form, must either be true or false for *all* concrete values represented by the variables.

The existing constraint solver, however, has the drawback that for technical reasons, constraints involving two variables are resolved by splitting one variable completely into concrete values. In the example, it would split $\boldsymbol{x}$ into the values $0, .., 255$ and only generate intervals $[0, 0]$, $[1, 1]$, $[2, 2]$ and $[3, 255]$ for $\boldsymbol{y}$.

B. Range Analysis

In this paper, we solve such constraints involving multiple variables by introducing a SAT solving technique to find suitable abstract values for both variables. To achieve this, we encode our constraints as SAT instances containing the bits of the variables involved as propositional variables. Therefore, the trace of instructions yielding to a guard is bit-blasted, which is detailed in the next section. For the example program, we would obtain the Boolean formula

$$f(\langle x_7, \ldots, x_0, y_7, \ldots, y_0 \rangle) := (\boldsymbol{x} + \boldsymbol{y} < 3)$$

at the conditional jump, i. e., for each solution of f, we have $(\boldsymbol{x} + \boldsymbol{y} < 3)$.

In a naïve approach, we would now call a SAT solver to a get solution $\langle b_0, \ldots, b_m \rangle$ with $b_i \in \{0, 1\}$ of f. A blocking clause $\neg b$ would then be added to suppress the old solution, and iteratively calling the SAT solver on $f \wedge \neg b$ would allow us to find all solutions of f. Since this approach involves solving 256 SAT instances for both variables, it is obviously infeasible.

If we set $x_7 = 1$, however, the formula $f \wedge x_7$ is no longer satisfiable, because this would imply $\boldsymbol{x} > 127$. This allows to infer that the lower bound of $\boldsymbol{x}$ is ≤ 127, which the key idea of the SAT based interval analysis: By testing whether the formula satisfiable setting the MSB to 0 or 1, the upper and lower bound of a variable can be confined. This step can be repeated with the next significant bit. Since this is similar to a binary search, the number of SAT instances to be solved is bound by the number of bits of the variables involved.

Using this techniques for the example program, the interval $[0, 2]$ would be inferred for both variables, i. e. the conditional branch can only be taken if the variables both lie in this interval. We can now restart the cycle with, say, $\boldsymbol{x} = [0, 2]$ and $\boldsymbol{y} = [3, 255]$. While in the latter case the conditional jump is never taken, the former case need to be refined further into the cases $\boldsymbol{x} = [0, 2]$ and $\boldsymbol{y} = [3, 255]$. In Fig. 2, these further refinements are shown as a tree. Input intervals that are still inconsistent for the conditional jump are marked with rounded corners. These input values are refined until the guard at the conditional jump is fulfilled (i. e., the jump is either taken or not taken for all values in the interval). Note also that for the state where $\boldsymbol{x} = \boldsymbol{y} = [0, 2]$, the SAT solving technique cannot find a proper refinement, so $\boldsymbol{x}$ is split into concrete values.

For all input intervals at leafs (marked with a rectangle), successors have to be generated, creating 7 successors in total for the example program using the new technique.

III. SAT Solving for Range Analysis

We will now detail the inner workings of the refinement step using SAT solving.

A. Generating a Boolean Constraint

Our CEGAR approach uses the following refinement loop, initially described by Kurshan [15]:

1) We store our refinements on a stack. In the first step, all variables are assigned to the most broad abstract value (full interval) of their domain.
2) A cycle of the PLC is executed using the abstract values in the variables.
3) If one of the above mentioned situations occurs, where the simulation cannot proceed, the constraint solver is used to find a new refinement for an input variable, which is then put on the stack and step 2 is repeated.
4) The atomic propositions of the formula are evaluated. If a truth value cannot be determined, again the constraint solver is used to find a new refinement, which is put on to the stack and step 2 is repeated.
5) The fresh successor state is stored in the state space.
6) The refinement on top of the stack is advanced until all values of the domain all values of the domain have been assigned. It is then removed. If the stack is empty all successors are created. Otherwise repeat with step 2.

In step 3 and 4, a constraint solver is called to transform a constraint on an intermediate value into a constraint on an input variable. This input variable is then assigned a refined value that avoid the inconsistent value for a decision after a restart of the cycle. To ensure termination, each refinement must create intervals that are proper subsets of the existing interval. If no proper subset is found, the input variable is split into concrete values, such that all constraints are fulfilled trivially.

To apply SAT solving to this process, we need to express the constraint as a Boolean formula. Since the execution trace of the program a straight trace from the beginning of the cycle to a guarded instruction, we can directly and automatically derive the Boolean formula by expressing all instructions on this trace by Boolean logic, in addition to the target constraint on the output variables. The force of this approach is that describing the relations semantics of a path in propositional Boolean logic is a standard technique in software model checking. The SAT solver thus resolves the relations between different bits automatically so as to infer input intervals. Expressing

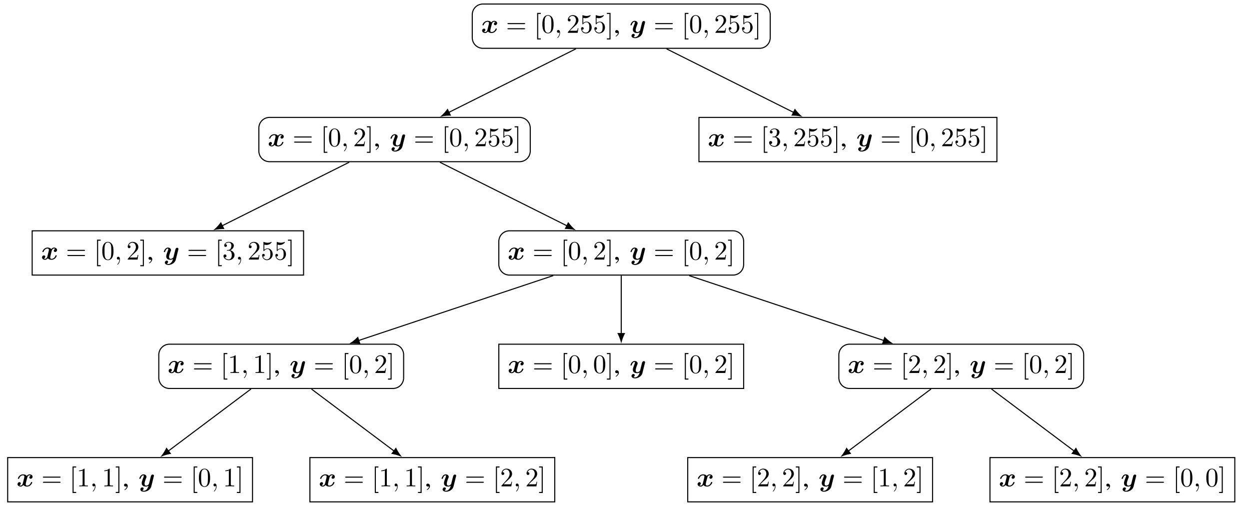

Fig. 2. Hierarchy of Refinements

the semantics of the instruction set, however, requires some careful engineering. Each operation needs to be described in Boolean logic. As an example, consider the operation `ADD x`, which adds the value of variable `x` to the current result in the accumulator, where it also stores the result. To bit-blast this instruction, introduce bit-vectors $\boldsymbol{x} = \langle \boldsymbol{x}_7, \dots, \boldsymbol{x}_0 \rangle$ for `x` as well as $\boldsymbol{cr} = \langle \boldsymbol{cr}_7, \dots, \boldsymbol{cr}_0 \rangle$ and $\boldsymbol{cr}' = \langle \boldsymbol{cr}'_7, \dots, \boldsymbol{cr}'_0 \rangle$ for the accumulator (current result) before and after the instruction. With additional intermediate carry-bits $\boldsymbol{c}$, the semantics of the operation is described propositionally as a full adder as follows:

$$\varphi(\boldsymbol{cr}, \boldsymbol{x}, \boldsymbol{cr}') = \begin{cases} (\bigwedge_{i=0}^{7} \boldsymbol{cr}'_i \leftrightarrow \boldsymbol{cr}_i \oplus \boldsymbol{x}_i \oplus \boldsymbol{c}_i) \wedge \neg \boldsymbol{c}_0 \wedge \\ (\bigwedge_{i=0}^{6} \boldsymbol{c}_{i+1} \leftrightarrow \\ (\boldsymbol{cr}_i \wedge \boldsymbol{x}_i) \vee (\boldsymbol{cr}_i \wedge \boldsymbol{c}_i) \vee (\boldsymbol{x}_i \wedge \boldsymbol{c}_i)) \end{cases}$$

Similar encodings can be derived for any operation available in IL. However, recall that SAT solvers typically expect input formulae to be in conjunctive normal form (CNF). We thus transform the formula into an equivalent formula in CNF [16]. Introducing fresh existentially quantified variables ensures that the size of the resulting formula is only a linear multiple of the size of $\varphi(\boldsymbol{cr}, \boldsymbol{x}, \boldsymbol{cr}')$. Observe that SSA conversion of the program makes bit-blasting trivial. For example, to bit-blast the fragment `ADD x; ADD y`, it is sufficient to pass $\varphi(\boldsymbol{cr}, \boldsymbol{x}, \boldsymbol{cr}') \wedge \varphi(\boldsymbol{cr}', \boldsymbol{y}, \boldsymbol{cr}'')$ converted into CNF to the SAT solver, where $\boldsymbol{cr}''$ denotes the bit-vector representing the output value of the accumulator after the second instruction.

B. Solving the Constraint using Nested Intervals

We will now detail how we use a SAT solver to infer the upper and lower bounds a variable $\boldsymbol{x} = \langle x_{n-1}, \dots, x_0 \rangle$ can take for a given constraint f'. In the first step, we add the interval to the constraint f' that $\boldsymbol{x}$ is already bound to. If $\boldsymbol{x}$ is bound to the interval $[a, b]$, we set

$$f := f' \wedge (\boldsymbol{x} \geq a) \wedge (\boldsymbol{x} \leq b).$$

This existing interval might have arisen in a previous refinement step or a previous execution of a cycle; thus we ensure here that these previous refinements are taken into account.

In the second step, we infer the lower bound $\boldsymbol{x}_l$ the variable $\boldsymbol{x}$ can take. To achieve this, we test whether $f \wedge \neg x_{n-1}$ is satisfiable. If so, we know that minimum $\boldsymbol{x}_l$ also has a 0 at this position. Otherwise, $f \wedge \neg x_{n-1}$ is unsatisfiable, so the minimum must be greater than 2^{n-1} and $\boldsymbol{x}_l$ has a 1 at the MSB position. This process is iterated for $x_{n-2}, x_{n-3}, \dots, x_0$ until all bits of $\boldsymbol{x}_l$ are inferred after n steps.

In the last step, the upper bound $\boldsymbol{x}_u$ of $\boldsymbol{x}$ is found. This is similar to the lower bounds, except we now start by testing whether $f \wedge x_{n-1}$ is satisfiable, which implies that the maximum $\boldsymbol{x}_u$ has a 1 at the MSB position. Otherwise, if $f \wedge x_{n-1}$ is unsatisfiable, the maximum must be smaller than 2^{n-1} which implies that the MSB of $\boldsymbol{x}_u$ is 0. This process can be iterated the same way.

This algorithm describes the case of an unsigned variable $\boldsymbol{x}$. For a signed variable, a similar algorithm could be used, that first deduces the sign of the minimum, testing if f has a solution with the MSB of $\boldsymbol{x}$ set.

C. Using the Negated Constraint

Consider the constraint $g := (\boldsymbol{x} + \boldsymbol{y} \geq 3)$. For g, the SAT based constraint solver will infer the interval $[0, 255]$ for $\boldsymbol{x}$ and $\boldsymbol{y}$, because the constraint is satisfiable for all values. Thus, the constraint solver will not find a proper refinement for $\boldsymbol{x}$ and $\boldsymbol{y}$, and it would still be necessary to split one variable into concrete values.

It this case, it is worthwhile to look at the negated constraint $\neg g$: Note, that $\neg g = f$ (from the last section), so the negation of g allows us to find a refinement as seen before. Therefore, our algorithm also takes the negated formula into account, when no proper refinement is found.

IV. Related Work

Our method is related to techniques from three fields of research, namely abstraction and refinement in model checking, the verification of software for PLCs, and SAT-based abstract

interpretation. The relations to previous contributions in either field are discussed in the remainder of this section.

A. Abstract Interpretation

The methodologies used to represent an abstract program semantics date back until the early days of abstract interpretation [3]. In particular, intervals were the first numerical abstract domain used in program analysis [3]. However, it took several decades until it was observed that combining bit- and word-length intervals using the reduced cardinal product allows to accurately reason about bit-manipulating programs [17], [18], [19]. In contrast to our work, these approaches do not apply any refinement to abstract descriptions. Following the observation of Regehr et al. [19] that low-level code requires accurate transformers there has been increasing interest in automatically synthesizing optimal abstractions [20], [21] from the concrete semantics of a program based on SAT solving [22], [23], [24]. Our method builds upon these methods.

B. Abstraction and Refinement

The idea of refining abstract representations of states using encountered over-approximations was formulated by Kurshan [15]. His observation led to the development of techniques for automated predicate abstraction [25] and the well-known CEGAR approach [5] which we implemented in our analysis framework. These techniques have found wide application in model checking in different contexts. For instance, Ball et al. [4] apply predicate abstraction and automated abstraction refinement to C code translated into Boolean programs [26]. In contrast, Henzinger et al. [6] propose a *lazy abstraction scheme* that refines only parts of the predicates in the program. Our refinement step for input variables can be seen as a simplified adaptation of their method. Furthermore, the abstraction-refinement scheme has found its way into all areas of model checkin such as bounded model checking [27]. Compared to our method, the main difference of existing techniques is that they operate on a general purpose abstraction of the program, such as Boolean programs [4], whereas our method exploits knowledge about the underlying hardware platform. It is also important to appreciate that backward interval analysis using SAT solving yields optimally precise initial constraints (w. r. t. the given constraints), thereby improving the precision of the first constraint-based implementation of our framework [10].

C. PLC Verification

Several attempts have been made in the past to apply model checking to software for PLCs. The first approach goes back to Moon [28], who translated programs given as Ladder Diagrams into the input language of SMV. This approach, however, only supports are very limited subset of Ladder Diagrams (namely, Boolean functions) and does not apply any abstraction, which leads to state explosion for small problems already. Later, Canet et al. [29] verified programs written in IL using NuSMV. The drawback of their method is that they only support a subset of IL and do not account for the cyclic scanning mode. A different approach was followed by Mertke and Frey [30], who translated IL programs into Petri nets, also not supporting the complete IL instruction set.

Huuck [31] used CADENCE SMV to verify PLC programs written as Sequential Function Charts (SFCs). Since parts of the defined SFC constructs have an ambiguous semantics, they only support a well-defined subset of the input language, which is described in [32]. In 2007, Pavlovic et al. [33] described an approach to translate PLC programs in Statement List — a vendor-specific language similar to IL — into the input language of NuSMV. Their approach, however, is not applicable to programs with several inputs without manual intervention. On the other hand, Süflow and Drechsler [34] applied equivalence checking using SAT to the task of PLC verification. Schlich et al. [9] introduced the concept of abstract simulation for PLC verification. This approach, which to a certain degree forms the basis of our framework, performs abstraction without refinement, and thus, often leads to spurious warnings. In [10], we described CEGAR for PLCs using a hand-written backward constraint solver. The main contribution of this paper with respect to that work is to replace the constraint solver with an automatic decision procedure to infer interval constraints. This approach alleviates the problem of designing backwards transformers, which can be challenging for certain kinds of operations.

V. CONCLUDING DISCUSSION

A. Conclusion

This paper describes an approach that combines model checking based on counterexample-guided abstraction refinement for PLCs with recent advances in automatic abstraction for programs whose semantics is described in terms of bit-vector relations. The key idea of this work is to integrate a description of the concrete (relational) semantics of a program using Boolean logic into the refinement loop. Describing the program in a relational fashion using Boolean encodings is a well-known technique, which is, for example, frequently used in bounded model checking [35]. The main contribution of our work is to pair this technique with successive refinement of interval abstractions based on SAT solving. The approach will thus directly profit from future advances in SAT solvers.

B. Future Work

Although we have illustrated the approach using SAT-based interval abstraction, the key steps of the algorithm — that is, encoding the concrete semantics of a program using propositional Boolean logic and deriving refined abstractions on the fly — are independent of the choice of abstract domain. It will therefore be interesting to evaluate the effectiveness and performance on weakly-relational domains such as octagons [36] by integrating the abstraction procedure recently developed by Brauer and King [24]. Their method rests on encodings, which are similar to those described in this work.

Orthogonal to choice of abstract domain is the question of integrating the reuse of refined constraints, which have been inferred for one cycle of the PLC, in the verification of another cycle. We believe that caching of such refinements has

the potential to significantly improve the performance of the described approach.

Acknowledgment

This work was supported by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89. Further, the work of Sebastian Biallas was supported by the DFG. The work of Jörg Brauer was, in part, supported by the DFG Research Training Group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* (AlgoSyn).

References

[1] International Electrotechnical Commission, *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems*. Geneva, Switzerland: International Electrotechnical Commission, 1998.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[3] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[4] T. Ball, B. Cook, S. Das, and S. K. Rajamani, "Refining approximations in software predicate abstraction," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 388–403.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 154–169.

[6] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM Press, 2002, pp. 58–70.

[7] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Principles of Programming Languages (POPL 78), Tucson, USA*. ACM, 1978, pp. 84–97.

[8] M. Karr, "Affine relationships among variables of a program," *Acta Informatica*, vol. 6, pp. 133–151, 1976.

[9] B. Schlich, J. Brauer, J. Wernerus, and S. Kowalewski, "Direct model checking of PLC programs in IL," in *Dependable Control of Discrete Systems (DCDS'09), Bari, Italy*, 2009.

[10] S. Biallas, J. Brauer, and S. Kowalewski, "Counterexample-guided abstraction refinement for PLCs," in *5th International Workshop on Systems Software Verification (SSV 2010), Vancouver, Canada*. Berkeley, CA, USA: USENIX Association, 2010, pp. 2–2.

[11] International Electrotechnical Commission, *IEC 61131-3: Programmable Controllers — Part 3 Programming languages*. Geneva, Switzerland: International Electrotechnical Commission, 1993.

[12] D. Kroening and O. Strichman, *Decision Procedures*. Springer, 2008.

[13] E. M. Clarke, *The Birth of Model Checking*. Berlin, Heidelberg: Springer-Verlag, 2008.

[14] 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 Trans. Program. Lang. Syst.*, pp. 451–590, 1991.

[15] R. P. Kurshan, *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton, NJ, USA: Princeton University Press, 1994.

[16] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, September 1986.

[17] J. Brauer, T. Noll, and B. Schlich, "Interval analysis of microcontroller code using abstract interpretation of hardware and software," in *SCOPES 2010*. ACM, 2010, to appear.

[18] J. Regehr and U. Duongsaa, "Deriving abstract transfer functions for analyzing embedded software," in *LCTES*. ACM, 2006, pp. 34–43.

[19] J. Regehr and A. Reid, "HOIST: A system for automatically deriving static analyzers for embedded systems," *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5, pp. 133–143, 2004.

[20] D. Monniaux, "Automatic Modular Abstractions for Linear Constraints," in *POPL*. ACM Press, 2009, pp. 140–151.

[21] T. Reps, M. Sagiv, and G. Yorsh, "Symbolic implementation of the best transformer," in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2004), Venice, Italy*, ser. LNCS, vol. 2937. Springer, 2004, pp. 252–266.

[22] J. Brauer and A. King, "Automatic abstraction for intervals using Boolean formulae," in *SAS*, ser. LNCS, vol. 6337. Springer, 2010, pp. 167–183.

[23] E. Barrett and A. King, "Range and set abstraction using SAT," *Electronic Notes in Theoretical Computer Science*, vol. 267, no. 1, pp. 17–27, 2010.

[24] J. Brauer and A. King, "Transfer function synthesis without quantifier eimination," in *ESOP*, ser. LNCS. Springer, 2011, to appear.

[25] R. Giacobazzi and F. Scozzari, "Intuitionistic implication in abstract interpretation," in *PLILP*, ser. LNCS, vol. 1292. Springer, 1997, pp. 175–189.

[26] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *SPIN*, ser. LNCS, vol. 1885. Springer, 2000, pp. 113–130.

[27] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[28] I. Moon, "Modeling programmable logic controllers for logic verification," *IEEE Control Systems Magazine*, vol. 14, no. 2, pp. 53–59, 1994.

[29] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in instruction list," in *2000 IEEE International Conference on Systems, Man, and Cybernetics, Nashville*, vol. 4. IEEE Computer Society Press, 2000, pp. 2449–2454.

[30] T. Mertke and G. Frey, "Formal verification of PLC-programs generated from signal interpreted petri nets," in *2001 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4. IEEE Computer Society Press, 2001, pp. 2700–2705.

[31] R. Huuck, "Software verification for programmable logic controllers," Dissertation, University of Kiel, Germany, April 2003.

[32] N. Bauer and R. Huuck, "A parameterized semantics for sequential function charts," in *Semantic Foundations Engineering Design Languages (SFEDL 2002)*, 2002, pp. 69–83.

[33] O. Pavlovic, R. Pinger, and M. Kollmann, "Automated formal verification of PLC programms written in IL," in *VERIFY*, ser. Workshop Proce., no. 259. CEUR-WS.org, 2007, pp. 152–163.

[34] A. Sülflow and R. Drechsler, "Verification of PLC programs using formal proof techniques," in *FORMS/FORMAT*. L'Harmattan, 2008, pp. 43–50.

[35] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Barcelona, Spain*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[36] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.