Noname manuscript No.
(will be inserted by the editor)

Leveraging Horn Clause Solving for Compositional Verification of PLC Software

Dimitri Bohlender · **Stefan Kowalewski**

Received: date / Accepted: date

Abstract Real-world PLC software is modular and composed of many different function blocks. Nevertheless, common approaches to PLC software verification do not leverage this but resort to inlining, or analyse instances of the same function block type independently.

With the advent of constrained Horn clauses as the basis for automated program verification, many state-of-the-art verification procedures build upon them. We illustrate how this formalism allows for a uniform characterisation of PLC program semantics and safety goals, derived from reactive systems safety foundations. Furthermore, we give a natural extension of the resulting encoding which enables compositional reasoning about modular software.

Due to the cyclic execution of PLCs, an engineer's mental model of a single function block often exhibits state machine semantics – partitioning a block's behaviour into different modes of operation. We illustrate how such a *mode space*, and similar high-level knowledge, can be integrated with our compositional characterisation.

We investigate the impact of each technique on the model checking performance by characterising PLC software verification problems, both in a non-compositional and a compositional way that may incorporate mode transitions, and solving them with an SMT solver. Evaluation of our prototypical implementation on examples from the PLCopen Safety library shows the effectiveness of both the chosen formalism and using high-level summaries.

Keywords Formal verification · Programmable logic controllers · Predicate abstraction · Constrained Horn clauses · Software safety · Industry automation

1 Introduction

Programmable logic controllers (PLCs) are devices commonly used for industrial automation tasks, e.g. monitoring the output flow of a tank and controlling according valves. Being

Dimitri Bohlender
E-mail: bohlender@embedded.rwth-aachen.de

Stefan Kowalewski
E-mail: kowalewski@embedded.rwth-aachen.de

Dimitri Bohlender · Stefan Kowalewski
Informatik 11 – Embedded Software, RWTH Aachen University, Aachen, Germany

used for repeatedly performing the same task, PLCs adopt a cyclic mode of execution, i.e. reading inputs (typically connected to sensors), executing a program, writing outputs (typically connected to actuators) and starting all over again. Thereby, PLCs are a widespread realisation of the theoretical concept of discrete event systems (DES), whose state changes on the observation of discrete events. The major difference is that, due to inputs being read only at the beginning of an execution cycle, non-simultaneous input changes may seem simultaneous to a PLC, but classical DES assumes events to occur in succession. While this does not affect the verification procedures, a specification for a DES may need adaptation if its semantics needs to be preserved in the context of PLCs.

PLC programs are usually written in the languages defined in the IEC 61131-3 standard. Using the concept of function blocks, consisting of an interface definition and a body containing the actual instructions that operate on this interface, programs can be modularised. In particular, a program is a function block and may, in turn, call other function blocks.

Since PLCs are tailored to industrial automation, they often operate in environments which require strong safety guarantees, and fixing bugs after commissioning entails significant costs. To approach these requirements, formal verification of the PLC software is desired and many successful applications of formal methods have been reported in the past. However most work operates on model level, analysing drafts and models of the system to be implemented, instead of the actual implementation (Ovatman et al., 2016). While such analyses are necessary to find conceptual problems early in the development cycle, they do not guarantee that the implementation will be free of bugs. Unfortunately, software level verification suffers from more limited scalability as it operates on more detailed and lower level semantics (Darvas et al., 2016). A similar argument can be made for the even more faithful but intractable hardware-level verification, which is why hardware-in-the-loop testing is complementary to software verification. While the core problems of software verification are undecidable and manifest themselves in the well known state space explosion problem, many real-world programs can nevertheless be analysed if suitable abstractions and procedures are utilised (Clarke et al., 2012).

In software that features several instances of the same component, or simply consists of many components, a common source of blowup of the state space is a non-compositional approach to verification. In such an approach there is no shared reasoning about different instances of the same function block type, and every instance is analysed independently. When PLC software verifiers, or their backends, do not support compositional reasoning but the programs of interest are modular, it is common to clone every function block's instructions for each of its instances, and potentially also *inline* all calls, that is replacing them by the callee's instructions. While these techniques get rid of the need for any compositional or inter-procedural reasoning, and work well for smaller examples, they entail an increase of the program size that is exponential in the function block nesting depth, and become problematic in bigger applications.

In PLC software verification it often pays off to perform preliminary static analyses or incorporate domain-specific information into the verification procedure to facilitate reasoning (Lange et al., 2013; Biallas et al., 2012). In contrast to general purpose software, in both PLC software and most DES formalisms, communication occurs in between executions of the program, or actions respectively. As a result, more complex tasks typically span several execution cycles and state has to be kept track of, such that subsequent cycles resume operating in the same mode. Accordingly, an engineer's mental model of a function block is often a state machine which exhibits different behaviours, depending on the current state, or *mode*, it is in. We observe that many function blocks indeed exhibit such state machine semantics, often featuring distinct modes for initialisation, errors and steps of computations.

Most prominently, even PLCopen, an organisation which specialises in technical specifications around the IEC 61131-3, uses state machines to partition and relate the possible behaviours of function blocks (PLCopen TC5, 2006).

1.1 Approach

Just as thinking of a block's semantics in terms of different modes reduces the complexity and helps engineers with reasoning on a higher level, so it may simplify reasoning for an automatic verification procedure. For example, since it encompasses global information on reachability of modes, it might help with avoiding analysis of behaviour that is irrelevant w.r.t. the property of interest. *Mode abstraction* is a static analysis that, given the variable which encodes a block's mode, computes its *mode space* – an abstract transition relation over the block's modes.

To maximise the benefit of having such a block summary during verification, the program semantics must be formalised in a way that allows for compositional reasoning, i.e. facts learned about a function block type should not have to be relearned for its instances. However, as observed in recent iterations of the annual *Competition on Software Verification* (SV-COMP) and our previous work on PLC software verification (Bohlender et al., 2018; Bohlender and Kowalewski, 2018b), classical model checking based on *binary decision diagrams* (BDDs) does not scale to software of relevant size, even though it can exploit compositionality to a certain extent. Instead, state-of-the-art verification procedures rely on *satisfiability modulo theories* (SMT) solving (Beyer, 2019). *Constrained Horn clauses* (CHCs) are both a particular class of SMT and a formalism for symbolic model checking, which has proved successful and enabled new software verifiers to outperform established tools (Beyer, 2015). To leverage these advances, we adopt CHCs to characterise PLC program semantics, high-level function block summaries, and safety goals in a uniform and compositional way.

1.2 Contributions

This work is an extended version of our conference paper (Bohlender and Kowalewski, 2018a), where we initially proposed mode abstraction and evaluated its impact on the model checking performance in a CHC-based verification pipeline.

In contrast to the conference paper, the primary contribution of this work is a comprehensible approach to CHC-based PLC software verification in general, derived from reactive systems safety foundations, and its extension to a compositional characterisation of program semantics that integrates with high-level summaries, like those provided by mode abstraction. To this end,

- we provide the first comprehensive exposition of how to achieve a CHC-based characterisation of PLC software safety, starting from state machine semantics,
- illustrate how this characterisation can be extended to enable compositional reasoning, and how high-level knowledge can be exploited, using the example of mode abstraction,
- evaluate how both the compositional characterisation and integration with mode abstraction affect the verification time on benchmarks of varying difficulty,
- provide all artefacts needed to study the characterisation and reproduce our results.

Note that besides the implementation of mode abstraction, all aspects have been reimplemented and extended to improve understandability. In particular, the compositional characterisation presented here arises naturally from the non-compositional one, the evaluation not

only investigates the effect of mode abstraction but also the compositional reasoning on its own, and the provided binaries also provide means to export several variants of characterisations of the benchmarks, for self-study and experimentation.

1.3 Related Work

The endeavour of verifying PLC software goes back to Moon (1994) who used the SMV formalism (McMillan, 1993) to characterise programs written in the Ladder Diagram programming language, starting off a line of research that seized on this approach (Ovatman et al., 2016), e.g. to specifiy and verify reusable components (Ljungkrantz et al., 2010). Although SMV targets hardware verification, and Ladder Diagram indeed is a circuit-like language without control flow, most present-day PLC software verifiers still use variants of SMV for model checking higher-level PLC programming languages (Darvas et al., 2016; Beckert et al., 2015). Model-level verification, in contrast, employs a variety of model checkers and formalisms, depending on the property of interest, e.g. time-based specifications require modelling of timed automata, while concurrency and interlocking are analysed with variants of Petri nets (Ovatman et al., 2016).

While different automata classes have previously been utilised as specifications, e.g. by Frey et al. (2012), our work is different in that it uses automaton-like high-level function block summaries to alleviate the complexity of verifying modular PLC software. While this idea is related to predicate abstraction based verification (Biallas et al., 2013; Graf and Saïdi, 1997) we employ a lightweight static analysis called *mode abstraction* to compute the abstraction, prior to running the actual verification procedure. This is possible since we do not treat all variables equally but focus on variables that encode a block's *mode*. This approach is in line with the common theme of performing preliminary static analyses, or incorporating domain-specific knowledge, prior to running a generic software verification procedure (Carter et al., 2016). In the overarching concept of CHC-based verification and exploiting high-level knowledge, mode abstraction only serves as an example, as other constraints, e.g. contracts (Quinton and Graf, 2008), can be integrated in a similar fashion.

The fact that not all variables of a program are used in the same way but exist for different purposes has been exploited in the past for choosing most suitable abstract domains (Apel et al., 2013). While approaches that construct state machines from behaviour traces seem related, they are unfit for summaries due to their under-approximating nature. Since the proposal of mode abstraction in our conference paper, it has also been seized on in test generation for PLC software (Simon and Kowalewski, 2018).

Since few work considers PLC programs that feature several instances of the same non-trivial function block, the need for a formalism that enables compositional reasoning rarely arises. Instead, popular options are inlining of calls at the expense of increasing the program's size (Bohlender et al., 2016; Lange et al., 2013), and cloning of the callee's body for every instance (Biallas et al., 2012; Darvas et al., 2013).

In line with this, we are not aware of any SMT-based PLC software verifier which makes use of inter-procedural, compositional reasoning. However, the CHC formalism has already been successfully applied in the context of compositional verification of general purpose software (Bjørner et al., 2015), and non-compositional model checking of PLC software (Bohlender et al., 2016). Starting from reactive systems safety foundations, this work provides a compositional characterisation of PLC software that differs from the one presented in our conference paper (Bohlender and Kowalewski, 2018a), being simpler and more flexible, but also more appropriate for the CHC solving backend (Komuravelli et al., 2015).

```
1  FUNCTION_BLOCK ReqHandler
2    VAR_INPUT  data    :WORD;          END_VAR
3    VAR        DiagCode:WORD;          END_VAR
4    VAR_OUTPUT res     :WORD;          END_VAR
5    // Body omitted ...
6  END_FUNCTION_BLOCK
7
8  PROGRAM Main
9    VAR_INPUT  req:BOOL; in:WORD;      END_VAR
10   VAR        m  :BOOL; h :ReqHandler; END_VAR
11   VAR_OUTPUT out:WORD;               END_VAR
12
13   // On rising edge, forward data to handler h
14   IF (req AND NOT m) THEN
15     h(data:=in, res=>out);
16   ELSE
17     h(res=>out);
18   END_IF
19   m:=req;
20 END_PROGRAM
```

Listing 1 Example program that delegates requests to a handler

Although we focus on reachability checking, the general approach is not restricted to it and can be adapted to more expressive logics (Beyene et al., 2016).

1.4 Outline

We commence with an example that motivates mode abstraction, illustrating why knowledge of a function block's mode space, or similar high-level knowledge, may simplify the verification of certain properties in a compositional setting. Subsequently, Section 3 recapitulates mode abstraction as a means to over-approximate a function block's state machine semantics, and the use of CHCs in the context of software verification.

The core contributions start with Section 4, where a CHC-based characterisation of PLC semantics is derived from reactive systems safety foundations, which is then extended in Section 5 to take advantage of the modularity of real-world PLC software. In Section 6 we present our experimental results regarding the feasibility of the proposed characterisations and conclude our work in Section 7.

2 Motivating Example

Consider the program from Listing 1, written in *Structured Text*, which implements a delegation of requests to a handler, and serves us as a running example throughout this paper. It features a main function block with two inputs `req` and `in`, for incoming requests and data respectively. In every cycle, it calls the nested `ReqHandler` instance `h` and writes its results to `out`. However, the input data `in` is only forwarded to `h` when a rising edge for `req` is detected, which indicates a new request in need of processing.

The actual implementation of `ReqHandler` might be arbitrary complex, but is not relevant for understanding the advantage of mode abstraction. It is therefore omitted in the

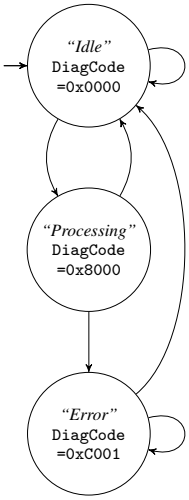

Fig. 1 Example mode space of `ReqHandler`

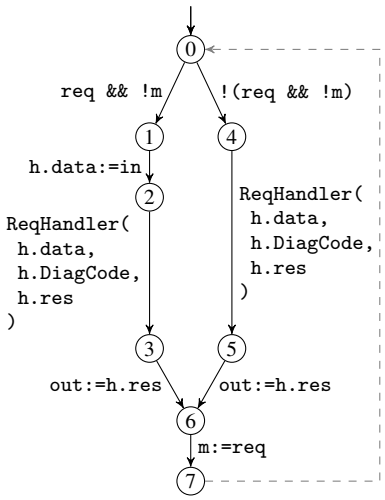

Fig. 2 Main CFA of the example program

listing, but not ignored during analysis or verification. The important detail is that it exhibits state machine semantics by using its `DiagCode` variable to track the current mode of operation. Among other things, its value encodes the current progress of handling the request and the errors occurred.

Assume that we are interested in checking (i) whether all requests are guaranteed to be processed in at most two execution cycles, or (ii) whether the processing may occur after an error, even without reentering a reinitialisation or idle mode.

Without auxiliary information about the behaviour of the `ReqHandler` block, the underlying verification procedure will have to derive all the necessary information on its own, possibly running into code whose precise semantics are irrelevant w.r.t. the property of interest and making the whole process unnecessary complex, potentially running out of time or memory. Although modern verification procedures do not explore the reachable state space naïvely but employ simplifications and abstraction refinement techniques, these are typically generic and are likely to be less effective than an abstraction which exploits domain specifics. This aspect grows more severe when a program under analysis features several instances of the same function block, but the verification procedure cannot reason about the instances in a unified way, instead resorting to exploring their state spaces independently.

In contrast, let Fig. 1 illustrate an example mode space of `ReqHandler`. Here, the vertices characterise sets of states that belong to the same mode, and the edges constrain how an execution of the block may change the mode. If we were to determine such an abstraction of the block's state space, we could analyse the properties of interest in a more directed way. For example, since there is no looping edge at the "Processing" mode, the block cannot stay in this mode for more than one PLC cycle. As a result, proving (i) that every request can be processed in at most two cycles amounts to proving the simpler property that, if there is a request, the "Processing" mode will be reached in a single cycle. Alternatively, if we wanted to prove (ii) that the request handler cannot reach the "Processing" mode directly from an "Error" mode but has to start anew from the "Idle" mode first, we would not even need to consult a verification procedure as the mode space already tells us that this is true.

3 Preliminaries

3.1 Program Representation

A PLC program consists of a set of (possibly nested) function block instances, which form stateful functions over their member variables. Conceptually they are similar to closures, or instances of classes from object-oriented programming with just one member function. During compilation, the state and functional parts are decoupled by *lowering* the function blocks to regular procedures with additional arguments for *references* to the state variables. As a result, PLC programs can be modelled as a set of procedures over a set of variables or memory locations.

We represent a procedure as a *Control Flow Automaton* (CFA) (Beyer et al., 2007) that distinguishes between the input and the remaining variables.

Definition 1 (Control Flow Automaton) A CFA $A = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_e, l_x)$ is a directed graph, where the vertices L are the program locations, with $l_e, l_x \subseteq L$ being the entry and exit locations respectively. The edges $E \subseteq L \times \mathit{Instr} \times L$ model the procedure's instructions, and their effect on control flow, over the vector of variables $\mathbf{X}$. If A is the program's main CFA, the procedure's input variables $\mathbf{X}_{in} \subseteq \mathbf{X}$ will be treated as the program's inputs.

Definition 2 (PLC Program) Following this, a PLC program is a pair $P = (M, \mathscr{A})$, where $\mathscr{A}$ is a set of CFAs and $M \in \mathscr{A}$ is the main CFA.

We restrict the presentation to a reduced set of instructions, featuring only *assignments*, *assumes* and *calls* – a common approach that is known to come without loss of generality.

Fig. 2 shows the main CFA of our example program from Listing 1. While the solid edges correspond to instructions, the dashed cycle-edge is not part of the CFA but was merely added for the sake of exposition, connecting the exit with the entry location and hinting at the cyclicity of the PLC. All of the instruction types occur in the example CFA. The boolean expressions at the outgoing edges of location 0 are assumes, representing the `if`-statement from line 14 of the example program. Assignments, like the `m:=req` at location 6, keep the notation from the source code (cf. line 19). Note that the calls' implicit input and output assignments have been lowered to regular assignments at locations $1, 3$ and 5, to allow for simpler call semantics. Furthermore, the calls now refer to a CFA `ReqHandler` which is derived from the corresponding function block type and operates on `h`'s member variables.

3.2 Mode Abstraction

Predicate abstraction (PA) was suggested by Graf and Saïdi (1997) as a technique for computing finite-state abstractions of possibly infinite state spaces, by grouping states that, given a set of predicates, satisfy the same subset of these predicates. For example, given the predicates $\{0 \leq x, x \leq 10\}$, where x is a variable of the considered system, PA partitions the state space into the states where $x \in [-\infty, -1]$, $x \in [0, 10]$ and $x \in [11, \infty]$. However in most applications the predicates yielding useful paritionings are not known beforehand and may be arbitrarily complex, requiring the use of computationally expensive SMT solving. Mode abstraction is a special case of PA where the predicates correspond to concrete valuations of *mode variables*, cf. Fig. 1, and can therefore be computed automatically with a lightweight static analysis. In the following we illustrate how mode abstraction can be implemented upon a standard value set analysis (VSA) for CFAs (Beyer et al., 2007; Lange et al., 2013).

Algorithm 1: ModeAbstraction$(P, \iota, modeVar)$

Input : program $P = (M, \mathscr{A})$
abstracted initial state $\iota : \mathbf{X}_M \to D$
identifying function $modeVar : CFA \to \mathbf{X}$
Variables: VSA result $\rho_M : CFA \to \mathbf{X} \to D$
mode transitions $\mu : CFA \to (\mathbb{Z} \to 2^{\mathbb{Z}})$

1 $\rho_M \leftarrow \mathsf{VSA}_{\mathsf{cycl}}(M, \iota)$ // Over-approximate observable values of all CFAs
2 $\mu \leftarrow \emptyset$ // Accumulates every CFA's mode transitions
3 **foreach** $A = (\mathbf{X}_A, \mathbf{X}_{in}, (L, E), l_e, l_x) \in \mathscr{A}$ **do**
4 $x_m \leftarrow modeVar(A)$ // Pick mode variable
5 $\mu_A \leftarrow \emptyset$ // Accumulates mode transitions of A
6 **foreach** $src \in \rho_M(A, x_m)$ **do** // Consider each mode as source
7 $\iota_A \leftarrow \rho_M(A)$ // Start from over-approximation
8 $\iota_A \leftarrow \iota_A[x_m \mapsto src]$ // with fixed source mode
9 $\iota_A \leftarrow \iota_A[x \mapsto D_\top \mid x \in X_{in}]$ // and unconstrained inputs.
10 $\rho_A \leftarrow \mathsf{VSA}(A, \iota_A)$ // Standard VSA of A
11 $\mu_A \leftarrow \mu_A \cup (src, \rho_A(A, x_m))$ // Add found target modes
12 $\mu \leftarrow \mu \cup (A, \mu_A)$
13 **return** μ

In contrast to concrete program executions, a VSA operates on values from an abstract domain D, such as intervals or sets of integers. Given an initial value $\iota(x) \in D$ for every variable x, it propagates and modifies values along the CFA's edges, considering nested CFAs, until a fixed point is reached. This resulting mapping ρ assigns an abstract value $\rho(l, x) \in D$ to every variable x at every location l, including locations in nested CFAs, and over-approximates the concrete values during any execution of the CFA that starts within the provided initial values.

In the context of PLCs, we only care about the values at each CFA's exit location, since the changing of state variables during the execution is not *observable*. Thus, it is more convenient to ignore the values at intermediate locations and treat ρ like a mapping from every variable x of a CFA A to the over-approximated observable value $\rho(A, x) \in D$.

However, since the plain VSA operates on CFAs and our program representation does not make the PLC cycle explicit, calling a VSA for some CFA A will only yield an over-approximation of the possible values after a single execution of A. To consider an arbitrary number of executions, an explicit edge from A's exit l_x to its entry l_e, which models the reading of inputs, could be added. Instead, to keep the notion simple, we let VSA_{cycle} denote a VSA which considers the implicit cycle-edge in the suggested way. With this in mind, mode abstraction can be understood as the combination of VSAs shown in Algorithm 1.

Before we can compute the mode spaces for each of the program's CFAs, we have to determine which modes exist in the first place. We do this in line 1 by invoking VSA_{cycle} on the program's main CFA M, to acquire a mapping ρ_M, which over-approximates the observable values of every CFA's variables. In the context of our running example, this may find that during any program execution, the handler's `DiagCode` variable may only take on values from the set $\rho_M(ReqHandler, DiagCode) = \{0, 0\text{x}8000, 0\text{xC}001\}$. The concrete values will form the over-approximated mode space's vertices, cf. Fig. 1.

In the next step, we iterate over the program's CFAs to compute the transitions between possible modes for each. The variables that implement mode semantics are typically known, e.g. `DiagCode` in our example, but could also be derived automatically from the context they occur in (Apel et al., 2013), e.g. they cannot be inputs, are only assigned constant values, and typically do not occur in arithmetic operations. In the pseudocode we model this knowledge through a function *modeVar*, which identifies a CFA's mode variable x_m (cf. line 4). We use

the results ρ_M of the program's VSA to determine the values x_m might take, and consider each of them as a concrete source mode *src* for the following nested VSAs in line 10.

The nested VSA's goal is to determine possible successor modes of *src*, after a single execution of the CFA A at any point in time. To this end, the nested VSA is initialised with A's over-approximated observable values $\rho_M(A)$, but x_m is initialised with mode *src* and each input is set to the largest element $D_\top$ of the abstract domain (cf. lines 7-9). Using non-deterministic inputs and reusing ρ_M, which encompasses behaviour that cannot be observed in a single CFA execution, guarantees safe over-approximation of mode transitions. In line 10, the result ρ_A of this CFA contains all the values x_m may take after a single execution starting in a particular mode *src*.

In the context of our running example, invoking this VSA on `ReqHandler`, initialising the mode variable `DiagCode` with the previously determined source mode 0x8000, we may find $\rho_{ReqHandler}(ReqHandler, DiagCode) = \{0, 0\text{xC001}\}$ to be the possible successor modes. This outcome forms the two outgoing edges from the vertex labelled with the predicate $DiagCode = 0\text{x}8000$ in Fig. 1, and of course depends on the actual implementation of `ReqHandler`. Performing such a VSA for every mode of a CFA yields an over-approximation of all mode transitions.

3.3 Constrained Horn Clauses in Software Verification

Hoare logic was proposed by Hoare (1969) as a formal system for reasoning about the correctness of programs. Later on, Blass and Gurevich (1987) made the case that existential positive least fixed-point logic provides a logical match for Hoare logic. Solving CHCs happens to correspond to this logic, and since standard proof rules from program verification readily correspond to Horn clauses, they have been widely adopted as a basis for program analysis (Bjørner et al., 2015).

In contrast to other formalisms and intermediate representations for program analysis, a CHC instance is just a particular SMT instance, rendering CHCs a very flexible formalism with access to SMT theories, e.g. enabling the modelling of non-trivial aspects like references and pointer arithmetic via the theory of arrays, or incremental checking of more complex specifications not feasible otherwise (Bohlender and Kowalewski, 2018b). In 2018, the growing interest in CHCs even spawned the dedicated CHC solving competition CHC-COMP, in addition to the renowned SV-COMP.

Definition 3 (Constrained Horn Clause) Given sets of variables $\mathscr{V}$, function symbols $\mathscr{F}$, and predicates $\mathscr{P}$, a CHC is a formula

$$\forall \mathscr{V}\ \underbrace{p_1(\mathbf{X}_1) \wedge \cdots \wedge p_k(\mathbf{X}_k) \wedge \varphi}_{\text{body}} \rightarrow h(\mathbf{X}),\ k \geq 0, \tag{1}$$

where φ is a constraint over $\mathscr{F}$ and $\mathscr{V}$, $\mathbf{X}_i, \mathbf{X} \subseteq \mathscr{V}$ are possibly empty vectors of variables, and $p_i(\mathbf{X}_i)$ is an application of a predicate p_i of arity $|\mathbf{X}_i|$.

We use *body* to refer to the antecedent of the CHC and *head* to denote h. A CHC is called a *query* if its head is free of $\mathscr{P}$ symbols and otherwise, it is called a *rule*. Following the convention of logic programming literature, we use the shorthand notation

$$h(\mathbf{X}) \leftarrow p_1(\mathbf{X}_1), \ldots, p_k(\mathbf{X}_k), \varphi. \tag{2}$$

A set of CHCs is satisfiable if there exists an interpretation of the predicates that satisfies each φ.

Intuitively, when characterising program semantics via CHCs, each p_i represents an unknown over-approximation of the values at a program location i, while a query defines a property to be proven, and each rule corresponds to a transition in the CFA and gives rise to a verification condition. Therefore, checking whether a program satisfies a safety property amounts to establishing the satisfiability of CHCs that encode the corresponding verification conditions, as shown in the following section.

4 Deriving a CHC-based Characterisation of PLC Software Safety

There are many ways to construct CHCs that characterise the checking of a partial correctness property in a program. Common approaches are formulating an operational semantics as an interpreter in a constraint logic program and specialise it for a given program (De Angelis et al., 2014), or direct translation to Horn logic using a weakest liberal precondition calculus (Bjørner et al., 2015). However, since we operate on the level of CFA, we approach this by extending the conceptually more related approach of characterising safety properties of a state machine, which the DES research community is also more familiar with.

A state machine, or hardware circuit, operates on a vector of state variables $\mathbf{X}$, whose values are possibly modified in every step of its execution. Let $I(\mathbf{X})$ be the constraint that expresses the set of initial states, and the possible steps be characterised by a *transition relation* $T(\mathbf{X},\mathbf{X}')$ which relates the variables' current and next values. Proving that all reachable states satisfy some constraint $safe(\mathbf{X})$ amounts to finding an inductive invariant $inv(\mathbf{X})$, such that the following CHCs are satisfied (Manna and Pnueli, 1995; Bjørner et al., 2015):

$$inv(\mathbf{X}) \leftarrow I(\mathbf{X}) \qquad inv(\mathbf{X}') \leftarrow inv(\mathbf{X}), T(\mathbf{X},\mathbf{X}') \qquad safe(\mathbf{X}) \leftarrow inv(\mathbf{X}) \tag{3}$$

Note that while inv is a predicate from $\mathscr{P}$, the others symbols are functions from $\mathscr{F}$.

4.1 Lifting of CHCs to PLC Software

Due to the cyclic execution of a PLC, the changing of state variables during the execution of a program is not observable by the environment. The only observable values are those which the variables take at the end of a PLC cycle, i.e. when the output variables provide the connected actuators with new values. Accordingly, and in contrast to state machines, specifications for PLC software typically refer to observable states only, and a safety constraint or invariant may be violated during program execution, as long as it still holds for the observable states.

Therefore, we cannot use the CHCs from (3) in their current form, but must adapt T to reflect the fact that specifications treat the whole PLC cycle like a single step:

$$cycle_x(\mathbf{X}) \leftarrow I(\mathbf{X}) \quad cycle_x(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), T_{cycle}(\mathbf{X},\mathbf{X}') \quad safe(\mathbf{X}) \leftarrow cycle_x(\mathbf{X}) \tag{4}$$

In this context, the predicate $cycle_x$ will correspond to the observable states at the exit of a PLC cycle, and T_{cycle} has to characterise the semantics of one *cycle-step* of the PLC. If, given a characterisation of initial values $I(\mathbf{X})$, these CHCs are satisfiable, then the found interpretation for the predicate $cycle_x$ over-approximates the program's observable states and proves that none of them violates the considered constraint $safe(\mathbf{X})$.

While the approach is sound, it is generally not possible to state T_{cycle} if the program contains loops – otherwise the halting problem were decidable. Therefore, the following sections focus on how additional CHCs can be used to characterise the semantics of a single execution of the program, without having to compute T_{cycle} explicitly.

4.2 Monolithic Program Semantics

In the following, let our program be given by $P = (M, \{M\})$, where $M = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_e, l_x)$ is the monolithic main CFA resulting from inlining all calls. To capture a PLC's cycle-step semantics T_{cycle} through a set of CHCs, these have to encompass the reading of new input values at the beginning of a cycle and the subsequent execution of the main CFA M.

4.2.1 Reading Inputs

We can make this sequence explicit by introducing another vector of variables $\mathbf{X}''$ and using

$$cycle_x(\mathbf{X}'') \leftarrow cycle_x(\mathbf{X}), \bigwedge_{x \in \mathbf{X} \setminus \mathbf{X}_{in}} x' = x, T_M(\mathbf{X}', \mathbf{X}'') \tag{5}$$

instead of the middle CHC from (4). Here, the reading of inputs is expressed by keeping the values of all non-input variables $\mathbf{X} \setminus \mathbf{X}_{in}$ while leaving the others unconstrained, and T_M relates this valuation $\mathbf{X}'$ at the beginning of the main CFA execution with the one at its end.

Alternatively, instead of introducing $\mathbf{X}''$, we can also express each step of this sequence in terms of separate CHCs

$$cycle_e(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), \bigwedge_{x \in \mathbf{X} \setminus \mathbf{X}_{in}} x' = x \qquad cycle_x(\mathbf{X}') \leftarrow cycle_e(\mathbf{X}), T_M(\mathbf{X}, \mathbf{X}') \tag{6}$$

by letting a new predicate $cycle_e$ characterise the intermediate values at the entry of the PLC cycle, i.e. right after reading the inputs. What remains to be determined is a characterisation of M which avoids the explicit computation of T_M.

4.2.2 CFA Semantics

Similar to the introduction of a new predicate to characterise the values at the beginning of a cycle, we can capture the semantics of M by introducing a predicate $M_i(\mathbf{X})$ for each $i \in L$, to characterise the possible valuations at every vertex of M.

In doing so, the characterisation $cycle_e$ of valuations at the beginning of the PLC cycle constrains the valuations at the entry of M, that is M_{l_e}, and the characterisation M_{l_x} of valuations at the exit of M constrains $cycle_x$. This link is expressed through the following rules:

$$M_{l_e}(\mathbf{X}) \leftarrow cycle_e(\mathbf{X}) \qquad cycle_x(\mathbf{X}) \leftarrow M_{l_x}(\mathbf{X}) \tag{7}$$

While M_{l_e} is constrained by $cycle_e$, the reachable valuations at other vertices of M depend on both the valuations at predecessor vertices and the connecting edges' instructions. Therefore, we acquire a characterisation $M_{l'}$ of valuations at location l' by creating corresponding rules for every edge $(l, instr, l') \in E$:

$$M_{l'}(\mathbf{X}') \leftarrow M_l(\mathbf{X}), T_{instr}(\mathbf{X}, \mathbf{X}') \tag{8}$$

where T_{instr} expresses the semantics of *instr* in an appropriate logic (cf. Section 4.2.3). Due to this inductive definition of the CFA semantics we avoid having to determine T_M explicitly.

Given a characterisation of initial values $I(\mathbf{X})$, the resulting set of CHCs needed for reasoning about a program's compliance w.r.t. a safety specification $safe(\mathbf{X})$, amounts to:

$$cycle_x(\mathbf{X}) \leftarrow I(\mathbf{X}) \tag{9}$$

$$cycle_e(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), \bigwedge_{x \in \mathbf{X} \setminus \mathbf{X}_{in}} x' = x \tag{10}$$

$$M_{l_e}(\mathbf{X}) \leftarrow cycle_e(\mathbf{X}) \tag{11}$$

$$M_{l'}(\mathbf{X}') \leftarrow M_l(\mathbf{X}), T_{instr}(\mathbf{X}, \mathbf{X}') \text{ for each } (l, instr, l') \in E \tag{12}$$

$$cycle_x(\mathbf{X}) \leftarrow M_{l_x}(\mathbf{X}) \tag{13}$$

$$safe(\mathbf{X}) \leftarrow cycle_x(\mathbf{X}) \tag{14}$$

Note that in the context of PLCs, we usually do not deal with a set of initial states but have a defined vector of initial values $\mathbf{X}_0$. Therefore, in practice, we use the fact $cycle_x(\mathbf{X}_0)$ instead of the more general CHC (9).

4.2.3 Instruction Semantics

The logical characterisation $T_{instr}(\mathbf{X}, \mathbf{X}')$ of both *assignments* and *assumes* can be derived from their standard *strongest postcondition* predicate transformers (Dijkstra and Schölten, 1990; Bohlender et al., 2018), i.e.

$$T_{v:=expr}(\mathbf{X}, \mathbf{X}') := (v' = expr \wedge \bigwedge_{x \in \mathbf{X}, x \neq v} x' = x) \tag{15}$$

$$T_{expr}(\mathbf{X}, \mathbf{X}') := (expr \wedge \bigwedge_{x \in \mathbf{X}} x' = x) \tag{16}$$

Note that the big conjunctions guarantee that assignments change only the value of v, and assumes do not modify any value but merely require an expression *expr* to hold.

For example, if $m := req$ from Fig. 2 is the instruction of interest, its characterisation over the sets $\mathbf{X}, \mathbf{X}'$ is

$$m' = req \wedge \bigwedge_{x \in \mathbf{X}, x \neq m} x' = x. \tag{17}$$

5 Compositional Characterisation

While the characterisation from Section 4 already gives access to the state-of-the-art verification procedures implemented in CHC solvers, it arises from a monolithic program representation. Such a non-compositional characterisation is devoid of the modularity of the original program, which can therefore not be exploited. In addition, the inlining may result in a significant increase of the program size, potentially causing a state space explosion during the following verification phase.

In the compositional setting, the program under analysis has call instructions and is therefore given by $P = (M, \mathscr{A})$, where $M = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_e, l_x)$, and each CFA $A \in \mathscr{A}$ corresponds to a function block type of the PLC program, i.e. not to a function block instance.

In the following sections, we present a compositional characterisation of the program semantics, continuing from the set of CHCs that we had constructed in Section 4.2.1:

$$cycle_x(\mathbf{X}) \leftarrow I(\mathbf{X}) \tag{18}$$

$$cycle_e(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), \bigwedge_{x \in \mathbf{X} \setminus \mathbf{X}_{in}} x' = x \tag{19}$$

$$cycle_x(\mathbf{X}') \leftarrow cycle_e(\mathbf{X}), T_M(\mathbf{X}, \mathbf{X}') \tag{20}$$

$$safe(\mathbf{X}) \leftarrow cycle_x(\mathbf{X}) \tag{21}$$

5.1 Concept

To enable truly compositional verification and efficient incorporation of function block summaries into CHC-based verification, the characterisation should reason about a CFA's semantics at function level, i.e. in terms of I/O-relations, and enable reuse of results in between analyses of different instances.

We approach this by extending the predicates M_i from Section 4.2.2, which characterise the possible valuations at program location i of the CFA M, to capture the possible valuations at that location if M had been entered with a particular valuation. That is, we introduce a predicate $A_i(\mathbf{X}_A, \mathbf{X}'_A)$ for every program location $i \in L_A$ of every CFA $A = (\mathbf{X}_A, \mathbf{X}_{A,in}, (L_A, E_A), l_{A,e}, l_{A,x})$, to express a valuation $\mathbf{X}'_A$ at location i that is transitively reachable when entering A with $\mathbf{X}_A$. In doing so, each $A_{l_{A,x}}(\mathbf{X}_A, \mathbf{X}'_A)$ represents an unknown over-approximate summary of a procedure, relating the states before and after a call of A, and each rule gives rise to a modular verification condition for one procedure (Komuravelli et al., 2015). Furthermore, such a predicate can be augmented with another summary which constrains the pre- and post-valuation, $\mathbf{X}_A$ and $\mathbf{X}'_A$ respectively (cf. Section 5.3).

In the following sections, we adapt and extend the non-compositional CHCs to make the predicates A_i express the intended semantics. Note that as a result, we will be able to replace $T_M(\mathbf{X}, \mathbf{X}')$ from (20) with $M_{l_x}(\mathbf{X}, \mathbf{X}')$ since their semantics will coincide.

5.2 CFA Semantics

In the non-compositional characterisation, $cycle_e$ constrained the predicate $M_{l_e}(\mathbf{X})$, which expressed the possible valuations the main CFA can be entered with (cf. Section 4.2.2). However, with $M_{l_e}(\mathbf{X}, \mathbf{X}')$ now being supposed to characterise the valuation $\mathbf{X}'$ at l_e that is reachable from $\mathbf{X}$ at l_e, $\mathbf{X}$ and $\mathbf{X}'$ have to be equal:

$$M_{l_e}(\mathbf{X}, \mathbf{X}) \leftarrow cycle_e(\mathbf{X}) \tag{22}$$

Having established this link, we can now start constraining the predicates M_i for the remaining program locations $i \in L$ through rules that inductively build upon it. Note that there is no need to lift the non-compositional rule for $cycle_x$, to link the variables' values at the exit of the main CFA and PLC cycle end, since this is already covered by (20). Since the semantics of every CFA A is formalised in the same way, we do not treat the case $A = M$ separately but refer to $A = (\mathbf{X}_A, \mathbf{X}_{A,in}, (L_A, E_A), l_{A,e}, l_{A,x})$ in the following.

With the non-compositional CHCs, a predicate for program location i was constrained by the predicates of predecessor locations and the connecting instructions. This is also the case in the compositional characterisation, yielding rules for every edge $(l', instr, l'') \in E_A$:

$$A_{l''}(\mathbf{X}_A, \mathbf{X}''_A) \leftarrow A_{l'}(\mathbf{X}_A, \mathbf{X}'_A), T_{instr}(\mathbf{X}'_A, \mathbf{X}''_A) \tag{23}$$

That is, if it is possible to reach a valuation expressed by $\mathbf{X}'_A$ at location l', when entering the CFA with $\mathbf{X}_A$, and an there is an instruction that leads from l' to location l'' that transforms the values into $\mathbf{X}''_A$, then $A_{l''}$ should relate the entry valuation $\mathbf{X}_A$ with $\mathbf{X}''_A$. Note that in the compositional setting, we must also provide the characterisation T_{instr} of a call instruction (cf. Section 5.3).

At this point, all of the relations $A_{l_{A,e}}(\mathbf{X},\mathbf{X}')$ would be empty, except for M_{l_e}, since there are no rules to derive appropriate $\mathbf{X}$ and $\mathbf{X}'$. For the main CFA M this is not an issue, since M_{l_e} is given through $cycle_e$, and the rules for the remaining M_i are based on this non-empty relation. So while we have characterised the intra-procedural semantics, we are still missing some inter-procedural semantics, i.e. the rules that constrain $A_{l_{A,e}}$ depending on where A is called from.

Due to the hierarchical nature of PLC software, if a function block instance b of type B is called, the callsite is within some parent function block instance a of a type A, and the memory locations of b are a subset of those of a. Accordingly, the valuation $\mathbf{X}'_b$ that can reach $l_{B,e}$ is a subset of the valuation $\mathbf{X}'_A$ at a callsite l_c, yielding a CHC similar to (22):

$$B_{l_{B,e}}\left(\mathbf{X}'_b,\mathbf{X}'_b\right) \leftarrow A_{l_c}(\mathbf{X}_A,\mathbf{X}'_A) \tag{24}$$

where $\mathbf{X}'_b \subseteq \mathbf{X}'_A$ are the primed variables of instance b.

Given a characterisation of initial values $I(X)$, the resulting set of CHCs needed for compositional reasoning about a program's compliance w.r.t. a safety specification $safe(\mathbf{X})$, amounts to

$$cycle_x(\mathbf{X}) \leftarrow I(\mathbf{X}) \tag{25}$$

$$cycle_e(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), \bigwedge_{x\in\mathbf{X}\setminus\mathbf{X}_{in}} x' = x \tag{26}$$

$$M_{l_e}(\mathbf{X},\mathbf{X}) \leftarrow cycle_e(\mathbf{X}) \tag{27}$$

$$cycle_x(\mathbf{X}') \leftarrow cycle_e(\mathbf{X}), M_{l_x}(\mathbf{X},\mathbf{X}') \tag{28}$$

$$safe(\mathbf{X}) \leftarrow cycle_x(\mathbf{X}) \tag{29}$$

and the addition of the following CHCs for each $A = (\mathbf{X}_A,\mathbf{X}_{A,in},(L_A,E_A),l_{A,e},l_{A,x}) \in \mathscr{A}$:

$$A_{l''}(\mathbf{X}_A,\mathbf{X}''_A) \leftarrow A_{l'}(\mathbf{X}_A,\mathbf{X}'_A), T_{instr}(\mathbf{X}'_A,\mathbf{X}''_A) \text{ for each } (l,instr,l') \in E_A \tag{30}$$

$$B_{l_{B,e}}\left(\mathbf{X}'_b,\mathbf{X}'_b\right) \leftarrow A_{l_c}(\mathbf{X}_A,\mathbf{X}'_A) \text{ for each call } (l_c,B(\mathbf{X}_b)),l') \in E_A \tag{31}$$

where $\mathbf{X}'_b \subseteq \mathbf{X}'_A$ and $B = (\mathbf{X}_B,\mathbf{X}_{B,in},(L_B,E_B),l_{B,e},l_{B,x}) \in \mathscr{A}$ is the called CFA. For clarity, a concrete instantiation of these CHCs for our running example is provided in Appendix A.

5.3 Call Semantics

The characterisation of *calls* is based on the predicates $B_{l_{B,x}}$ that we introduce for every CFA B. Given a call $B(\mathbf{arg})$ to a CFA B with the variables $\mathbf{arg}$ to operate on, its characterisation within a parent CFA A is given by

$$T_{B(\mathbf{arg})}\left(\mathbf{X_A},\mathbf{X}'_A\right) := B_{l_{B,x}}(\mathbf{arg},\mathbf{arg}') \wedge \bigwedge_{x\in\mathbf{X}_A\setminus\mathbf{arg}} x' = x \wedge S_B(\mathbf{arg},\mathbf{arg}') \tag{32}$$

where $\mathbf{arg} \subseteq \mathbf{X}_A$, and the predicate $B_{l_{B,x}}$ captures which output valuation $\mathbf{arg}'$ can be reached given the input $\mathbf{arg}$. The big conjunction guarantees that all variables of $\mathbf{X}_A$ that are not used

as arguments keep their values – they are not visible to the callee. This is also the place where additional information about the callee can be exploited, by restricting the behaviour that needs to be analysed by the underlying solver, through a constraint $S_B(arg, arg')$.

To get a more intuitive understanding of the characterisation of calls, consider the call `ReqHandler(h.data, h.DiagCode, h.res)` from our running example (Fig. 2). The call occurs in the context of the main CFA, so

$$\mathbf{X} = (req, in, m, h.data, h.DiagCode, h.res, out), \tag{33}$$

and $\mathbf{X}'$ contains the primed variants of these variables. Given the callee's exit location $l_{ReqHandler,x}$, the call can be characterised as follows, if no summary is available:

$$\begin{aligned} &ReqHandler_{l_{ReqHandler,x}}(h.data, h.DiagCode, h.res, \\ &\qquad h.data', h.DiagCode', h.res') \\ &\wedge\ req' = req \wedge in = in' \wedge m' = m \wedge out' = out \end{aligned} \tag{34}$$

In contrast, if we wanted to exploit our knowledge of the mode space (cf. Fig. 1), we would have provided the CHC solving procedure with the following summarising constraint that expresses all mode transitions:

$$\begin{aligned} &(h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ &\qquad \vee\ h.DiagCode' = \text{0x8000}) \\ &\wedge (h.DiagCode = \text{0x8000} \rightarrow h.DiagCode' = 0 \\ &\qquad \vee\ h.DiagCode' = \text{0xC001}) \\ &\wedge (h.DiagCode = \text{0xC001} \rightarrow h.DiagCode' = 0 \\ &\qquad \vee\ h.DiagCode' = \text{0xC001}) \end{aligned} \tag{35}$$

For example, if a `ReqHandler` instance is in the mode characterised by $DiagCode = 0$, $h.DiagCode' = 0 \vee h.DiagCode' = \text{0x8000}$ encodes that calling this instance will cause it to either stay in that mode, or end up in the one characterised by $h.DiagCode' = \text{0x8000}$.

6 Experiments

6.1 Implementation

We implemented Java-prototypes of mode abstraction and both the non-compositional and the compositional approach to CHC-based verification, using the publicly available SMT solver Z3 (de Moura and Bjørner, 2008) and the ARCADE.PLC platform for analysis of PLC software (Biallas et al., 2012). While ARCADE.PLC itself does not implement methods for CHC-based verification, we use its compiler frontend to acquire CFAs of PLC programs written in *Structured Text*, and build upon its VSA to implement mode abstraction. In contrast to the conference paper, the employed version of Z3 (v4.8.1) defaults to the SPACER-engine (Komuravelli et al., 2015) for solving CHCs, superseding PDR (Eén et al., 2011; Hoder and Bjørner, 2012), and is tailored to compositional reasoning.

Unlike the presented characterisation of single instructions, our implementation follows the common approach of characterising *large-blocks* of instructions (Beyer et al., 2009). However, for self-study and experimentation, the supplied artefacts allow for generation of CHCs in various block encodings.

6.2 Benchmarks

PLCopen is an organisation which drives standardisation and technical specifications in automation. The *PLCopen Safety* library is a collection of such specifications. We experimented with two groups of PLC programs from this library. Programs from (PLCopen TC5, 2006) are single modules implementing particular safety concepts, while (PLCopen TC5, 2008) features user examples which combine these into larger applications. Most of these programs, or their components, exhibit mode semantics through `DiagCode` variables and address domain-specific problems, e.g. how to realise a safe emergency shutdown.

Since publication of the conference paper, the number of compositional programs and specifications has increased, such that the new benchmark subsumes the old one, and the sizes of these programs range from 120 to about 1500 program locations in the main CFA with inlined calls. The (invariant) specifications we use were either already formulated by PLCopen or derived from their technical specifications.

All experiments were performed on a 64 bit Linux machine with 3 GHz, 8 GB of RAM and a timeout of 1000 s. They can be reproduced with the artefacts available on our website[1].

6.3 Discussion

In the following, we discuss the results of our experiments with the 64 verification tasks, of which 23 concerned the elementary modules from (PLCopen TC5, 2006), while the remaining 41 referred to composite applications from (PLCopen TC5, 2008).

6.3.1 Non-compositional vs. Compositional Reasoning

Fig. 3 illustrates how much time the solving of both the non-compositional and the compositional CHCs took for each verification task. To facilitate the analysis of the effects of using a compositional characterisation, every data point corresponds to one verification task and its coordinates put the execution times of both approaches in relation. As a result, a data point on the line through the origin means that using the compositional formulation has no impact, and the dashed lines correspond to a change by an order of magnitude. Note that the time for generating the CHCs is negligible and factored out in the experiments, i.e. to allow for a direct comparison we only measure the runtime of the CHC solver.

We find that even with the non-compositional characterisation, Z3 already manages to to solve 38 verification tasks, which in particular subsume all tasks on elementary modules. As expected, *all* 26 timeouts occur for specifications concerning the composite applications. Despite the timeouts, this already outperforms previous approaches based on CEGAR and PA (Biallas et al., 2012, 2013), which struggle even with some of the elementary modules, e.g. PLCopen's `SF_MutingSeq`.

While classical model checking based on decision diagrams is common in the DES community, and a good choice for model-level verification (Ovatman et al., 2016), it is not competitive with solving the non-compositional characterisation of our verification tasks. As explicated in Section 1.1, experience shows that decision diagrams rarely scale to the problems arising in software verification, and only work well for restricted classes of programs, e.g. where "variables are only used in equality expressions [..] and not with other arithmetic operators" (Beyer and Stahlbauer, 2014). Our previous work on an overlapping

[1] `https://arcade.embedded.rwth-aachen.de/jdeds18_chc.tar.gz`

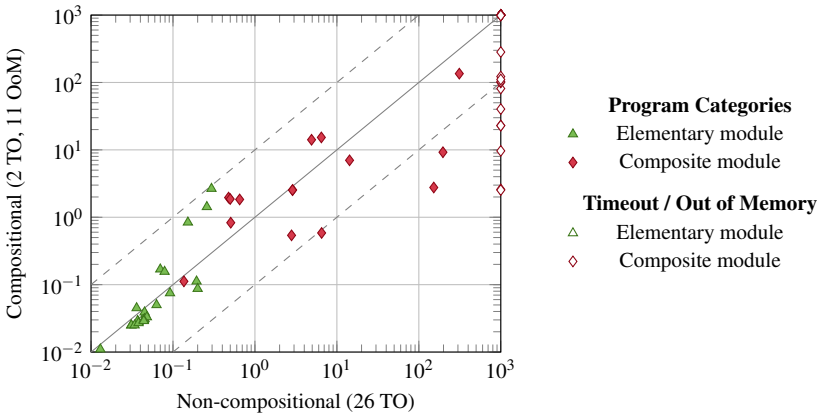

Fig. 3 Time [s] spent on verification of each task, using non-compositional and compositional reasoning

set of programs confirms this (Bohlender and Kowalewski, 2018b; Bohlender et al., 2018), finding BDDs to perform worse than SMT-based alternatives, only managing to explore the state space of few elementary modules, and running out of resources most of the time.

A direct comparison with the SMV-based verification of Darvas et al. (2016) is difficult, as their tool is not publicly available. However, they rely on NUXMV (Cavada et al., 2014), which is the latest symbolic model checker for the SMV formalism, and publicly available for academic purposes. Besides BDD-based reasoning, which is not sufficient for our verification tasks, it also implements PDR for checking reachability properties. Given that Z3's SPACER is an extension of PDR, it is not surprising that reachability checking for a SMV-based characterisation was found to perform similarly to satisfiability checking of our non-compositional characterisation (Bohlender and Kowalewski, 2018b). Therefore, an SMV-based approach like (Darvas et al., 2016), will at best perform comparable.

As soon as the properties of interest become more complex, even NUXMV will resort to the unfeasible BDDs. In contrast, CHCs are just a class of SMT, so more generic but still feasible procedures for the property of interest may exist, e.g. to express restart robustness (Bohlender and Kowalewski, 2018b). Furthermore, since NUXMV is the only modern model checker for SMV, but a closed-source standalone tool, it is unfeasible to express uncommon specifications in SMV or build competitive procedures upon NUXMV's. Both was found to be practical with CHCs (Bohlender and Kowalewski, 2018b; Bohlender et al., 2016).

Let us now return to Fig. 3, and consider the data points' y-coordinates, which correspond to the measured CPU times for solving each task's compositional characterisation, using the same engine and parameters as for the non-compositional one. For the simpler tasks, that take less than 3 s to analyse, there is no significant difference. If any, the non-compositional characterisation is slightly faster for some of them, since the overhead that the compositional solving introduces is not amortised that quickly.

However for the modular and more complex programs, compositional reasoning yields improvements by more than an order of magnitude. In line with this, the approach also halves the number of problematic instances, running out of resources only 13 times. Note though that, unlike in the non-compositional case, only 2 of these are timeouts. The running out of memory is due to SPACER being tailored towards compositional reasoning by creating and checking local reachability queries for individual procedures, and also maintaining both

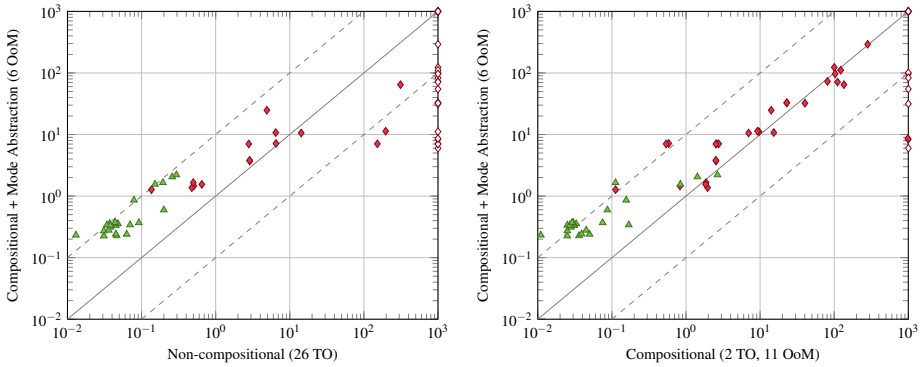

Fig. 4 Comparison of time [s] spent checking properties with compositional encoding and mode abstraction

may- and must-summaries for them, which requires more memory than non-compositional reasoning.

6.3.2 Compositional Reasoning with Mode Abstraction

Similar to the diagram in Fig. 3, the plots in Fig. 4 visualise the impact of performing mode abstraction, and solving the accordingly augmented CHCs, on the total verification time.

The first plot compares non-compositional reasoning with solving summary-augmented CHCs. The experimental results look similar to Fig. 3 but with all data points shifted in favour of the non-compositional approach. This shift originates in the time needed to compute the mode abstraction, and while resulting in a significant decrease of the overall verification performance for small or non-modular programs, this overhead becomes negligible for the more complex ones. However based on this diagram alone, it is difficult to judge whether the incorporation of mode abstraction pays off, or results in similar verification performance.

To allow for a direct comparison, the second plot in Fig. 4 relates both compositional approaches. As in the previous plot, we can see that the overhead becomes negligible once the verification tasks become so complex that they cannot be solved within 10 s, and the clustering around the line through the origin shows that most of the considered specifications do not profit significantly from the information that mode abstraction provides.

However we also see that in some cases, mode abstraction reduces the runtime by more than an order of magnitude, even enabling the checking of previously problematic properties. The combination of compositional reasoning and mode abstraction manages to halve the number of cases where the verification procedure runs out of resources once more, leaving only six verification tasks unsolved.

Overall we observe that the overhead of performing mode abstraction is insignificant w.r.t. the time needed for solving the resulting CHCs, while potentially improving the performance significantly when used in the context of more complex, composite applications.

Since our experiments aimed to demonstrate the feasibility of using CHCs, and illustrate the impact of using different characterisations, we were less interested in the qualitative results, i.e. whether the implementations comply with the specifications. Nevertheless, the benchmark covered all possible outcomes, as we found that of the 58 specifications that could be checked, 40 were proven to have been implemented correctly, and in 18 cases violations were found. Our experiments also suggest that proving safety, i.e. finding a solution

to the CHCs, tends to be more difficult than finding counterexamples, as we measured the slowest proof of violation to occur at the 33 s mark, while proofs of safety took up to 292 s. Beyond that, no patterns were observed, and the measured times for proofs of safety were found to range over all orders of magnitude.

7 Conclusion

In recent years, CHCs have been widely adopted in the software verification community as a basis for program analysis and symbolic model checking. Since proof rules from program verification readily correspond to Horn clauses, many state-of-the-art procedures build upon them, and being just a particular class of SMT, CHCs form a flexible formalism with access to SMT theories.

To the best of our knowledge, the PLC verification community has not yet taken advantage of this development, as classical model checking procedures based on decision diagrams are in fact a good choice for DES, and verification at an abstract model-level. While these methods may also be applicable to restricted classes of programs, the common approaches to PLC software verification struggle with the considered PLC software, which is modular and composed of many function block instances.

Therefore, we provided a comprehensive exposition of how a CHC-based characterisation of PLC software safety can be derived from reactive systems safety foundations. While the basic characterisation is devoid of the modularity of PLC software, we found that leveraging CHC solving to perform the symbolic model checking is competitive with state of the art in PLC software verification. The non-compositional characterisation forms a simple but capable basis for extensions that are not feasible via common methods, e.g. verification of restart robustness (Bohlender and Kowalewski, 2018b).

To also exploit the modularity present in PLC software, the basic characterisation was extended to decouple function and data, and enable compositional reasoning during verification, by treating function blocks in terms of relations between entry and exit states. The resulting characterisation naturally integrates with additional information about a function block's behaviour, such as mode spaces, which may improve the verification performance.

While our results suggest that both compositional reasoning and integration of high-level information pay off, some verification tasks remain problematic, and in need of more elaborate approaches. To this end, it seems worthwhile to integrate our characterisation with auxiliary information from static analyses, and other approaches to simplify the CHCs in preprocessing (Bjørner et al., 2015), to alleviate the burden on the CHC solver.

A Compositional Characterisation of the Running Example

In the following we explicitly instantiate the characterisation from Section 5, that is clauses (25)-(31), using the example CFA from Fig. 2, including the higher-level knowledge provided by the mode abstraction (cf. Fig. 1). Note that for the sake of exposition, and lack of bit-operations in our example, we do not resort to the theory of fixed-size bit-vectors but capture the program semantics over booleans and integers. A characterisation over bit-vectors can and has been used in previous work (Bohlender and Kowalewski, 2018b).

Let $h.res = out$ be an example invariant the program should be compliant with, stating that the outputs of both function blocks coincide in all observable states. The characterisation can be split into three parts:

1. the PLC cycle, calling the program's main function block, is characterised by clauses (25) to (28),
2. the query for safety, characterised by clause (29),
3. and each CFA's characterisation, as given by clauses (30) and (31).

Similar to Equation 33, the variables of the occurring CFA instances are given by

$$\mathbf{X} = (req, in, m, h.data, h.DiagCode, h.res, out) \qquad \mathbf{X}_h = (h.data, h.DiagCode, h.res) \tag{36}$$

and $\mathbf{X}', \mathbf{X}'_h$, or $\mathbf{X}'', \mathbf{X}''_h$, are further variants of these variables that are primed once or twice, respectively, i.e.

$$\mathbf{X}' = \left(req', in', m', h.data', h.DiagCode', h.res', out'\right) \qquad \mathbf{X}'_h = \left(h.data', h.DiagCode', h.res'\right) \tag{37}$$

A.1 PLC Cycle

Since Listing 1 provides no explicit initialisation, all variables are initialised to 0 or *false* (cf. IEC 61131-3). Instantiating clauses 25-28 yields the following constraints:

$$cycle_x(\mathbf{X}) \leftarrow \begin{pmatrix} req = false \wedge in = 0 \wedge m = false \wedge h.data = 0 \wedge h.DiagCode = 0 \\ \wedge\ h.res = 0 \wedge h.out = 0 \end{pmatrix} \tag{38}$$

$$cycle_e(\mathbf{X}') \leftarrow cycle_x(\mathbf{X}), \begin{pmatrix} m' = m \wedge h.data' = h.data \wedge h.DiagCode' = h.DiagCode \\ \wedge\ h.res' = h.res \wedge h.out' = h.out \end{pmatrix} \tag{39}$$

$$Main_0(\mathbf{X}, \mathbf{X}) \leftarrow cycle_e(\mathbf{X}) \tag{40}$$

$$cycle_x(\mathbf{X}') \leftarrow cycle_e(\mathbf{X}), Main_7(\mathbf{X}, \mathbf{X}') \tag{41}$$

Here, 7 is the exit location of the main function block *Main*, and the predicate $Main_7(\mathbf{X}, \mathbf{X}')$ characterises the possible values $\mathbf{X}'$ after a single execution of the main CFA when started with values $\mathbf{X}$.

A.2 Query

Instantiating clause 29, we require all observable states to satisfy $h.res = out$:

$$h.res = out \leftarrow cycle_x(\mathbf{X}) \tag{42}$$

A.3 CFA Semantics

While the running example states an implementation of the main function block `Main`, no concrete implementation of the `ReqHandler` block is provided, but only its interface. Accordingly, no sensible characterisation of its body can be provided here, but should be easy to establish given a concrete CFA. Every CFA is characterised *independently* by instantiating clause (30) for every edge of the CFA, and (31) for every *call*-edge. For reference, complete characterisations of every benchmark are provided with the artefacts of this work.

In the following we characterise `Main` (cf. Fig. 2) by sequentially instantiating (30) for its *assume*-, *assign*-, and *call*-edges. The instantiation for the assumes $(0, req \wedge \neg m, 1)$ and $(0, \neg(req \wedge \neg m), 4)$ yields:

$$Main_1(\mathbf{X}, \mathbf{X}'') \leftarrow Main_0(\mathbf{X}, \mathbf{X}'), req' \wedge \neg m' \wedge \mathbf{X}'' = \mathbf{X}' \tag{43}$$

$$Main_4(\mathbf{X}, \mathbf{X}'') \leftarrow Main_0(\mathbf{X}, \mathbf{X}'), \neg\left(req' \wedge \neg m'\right) \wedge \mathbf{X}'' = \mathbf{X}' \tag{44}$$

and the assignments $(1, h.data := in, 2), (3, out := h.res, 6), (5, out := h.res, 6)$ and $(6, m := req, 7)$ become:

$$Main_2(\mathbf{X}, \mathbf{X}'') \leftarrow Main_1(\mathbf{X}, \mathbf{X}'), h.data'' = in' \wedge \begin{pmatrix} req'' = req' \wedge in'' = in' \wedge m'' = m' \\ \wedge\, h.DiagCode'' = h.DiagCode' \\ \wedge\, h.res'' = h.res' \wedge h.out'' = h.out' \end{pmatrix} \tag{45}$$

$$Main_6(\mathbf{X}, \mathbf{X}'') \leftarrow Main_3(\mathbf{X}, \mathbf{X}'), out'' = h.res' \wedge \begin{pmatrix} req'' = req' \wedge in'' = in' \\ \wedge\, m'' = m' \wedge h.data'' = h.data' \\ \wedge\, h.DiagCode'' = h.DiagCode' \wedge h.res'' = h.res' \end{pmatrix} \tag{46}$$

$$Main_6(\mathbf{X}, \mathbf{X}'') \leftarrow Main_5(\mathbf{X}, \mathbf{X}'), out'' = h.res' \wedge \begin{pmatrix} req'' = req' \wedge in'' = in' \\ \wedge\, m'' = m' \wedge h.data'' = h.data' \\ \wedge\, h.DiagCode'' = h.DiagCode' \wedge h.res'' = h.res' \end{pmatrix} \tag{47}$$

$$Main_7(\mathbf{X}, \mathbf{X}'') \leftarrow Main_6(\mathbf{X}, \mathbf{X}'), m'' = req' \wedge \begin{pmatrix} req'' = req' \wedge in'' = in' \\ \wedge\, h.data'' = h.data' \wedge h.DiagCode'' = h.DiagCode' \\ \wedge\, h.res'' = h.res' \wedge h.out'' = h.out' \end{pmatrix} \tag{48}$$

As stated in Section 5.3, the effect of calling a CFA B is captured by the predicate $B_{l_{B,x}}$, where $l_{B,x}$ is the exit location of B, similar to the embedding of `Main` in the PLC cycle in (41). However, since no concrete implementation of `ReqHandler` was stated, in the following, we use example locations 0 and 42 as entry and exit respectively. The instantiation of (30) for the calls $(2, ReqHandler(h.data, h.DiagCode, h.res), 3)$ and $(4, ReqHandler(h.data, h.DiagCode, h.res), 5)$ results in:

$$Main_3(\mathbf{X}, \mathbf{X}'') \leftarrow Main_2(\mathbf{X}, \mathbf{X}'), \begin{pmatrix} ReqHandler_{42}(h.data', h.DiagCode', h.res', \\ \qquad h.data'', h.DiagCode'', h.res'') \\ \wedge\, req'' = req' \wedge in'' = in' \wedge m'' = m' \wedge out'' = out' \\ \wedge\, (h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{x}8000) \\ \wedge\, (h.DiagCode = 0\text{x}8000 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{xC}001) \\ \wedge\, (h.DiagCode = 0\text{xC}001 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{xC}001) \end{pmatrix} \tag{49}$$

$$Main_5(\mathbf{X}, \mathbf{X}'') \leftarrow Main_4(\mathbf{X}, \mathbf{X}'), \begin{pmatrix} ReqHandler_{42}(h.data', h.DiagCode', h.res', \\ \qquad h.data'', h.DiagCode'', h.res'') \\ \wedge\, req'' = req' \wedge in'' = in' \wedge m'' = m' \wedge out'' = out' \\ \wedge\, (h.DiagCode = 0 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{x}8000) \\ \wedge\, (h.DiagCode = 0\text{x}8000 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{xC}001) \\ \wedge\, (h.DiagCode = 0\text{xC}001 \rightarrow h.DiagCode' = 0 \\ \qquad \vee\, h.DiagCode' = 0\text{xC}001) \end{pmatrix} \tag{50}$$

where the grey constraint is the optional information provided by the example mode space (cf. Section 5.3). Similar to the passing of state to `Main` in (40), the instantiation of clause (31) for the call edges yields:

$$ReqHandler_0(\mathbf{X}'_h, \mathbf{X}'_h) \leftarrow Main_2(\mathbf{X}, \mathbf{X}') \tag{51}$$

$$ReqHandler_0(\mathbf{X}'_h, \mathbf{X}'_h) \leftarrow Main_4(\mathbf{X}, \mathbf{X}') \tag{52}$$

This achieves the passing of `h`'s state to the functional characterisation of `ReqHandler` at both locations 2 and 4, such that the corresponding relations $ReqHandler_{42}$ in (49) and (50) are guaranteed to not be empty. Just as $Main_7(\mathbf{X}, \mathbf{X}')$ transitively depends on $Main_0(\mathbf{X}, \mathbf{X}')$, the exit location predicate $ReqHandler_{42}$ will transitively depend on the entry location predicate $ReqHandler_0$ for any concrete implementation of `ReqHandler`.

References

Apel S, Beyer D, Friedberger K, Raimondi F, von Rhein A (2013) Domain types: Abstract-domain selection based on variable usage. In: Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings, pp 262–278

Beckert B, Ulbrich M, Vogel-Heuser B, Weigl A (2015) Regression verification for programmable logic controller software. In: Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings, pp 234–251

Beyene TA, Popeea C, Rybalchenko A (2016) Efficient CTL verification via horn constraints solving. In: Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016., pp 1–14

Beyer D (2015) Software verification and verifiable witnesses - (report on SV-COMP 2015). In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, pp 401–416

Beyer D (2019) Automatic verification of C and java programs: SV-COMP 2019. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III, pp 133–155

Beyer D, Stahlbauer A (2014) Bdd-based software verification. International Journal on Software Tools for Technology Transfer 16(5):507–518

Beyer D, Henzinger TA, Théoduloz G (2007) Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, pp 504–518

Beyer D, Cimatti A, Griggio A, Keremoglu ME, Sebastiani R (2009) Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA, pp 25–32

Biallas S, Brauer J, Kowalewski S (2012) Arcade.plc: a verification platform for programmable logic controllers. In: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, pp 338–341

Biallas S, Giacobbe M, Kowalewski S (2013) Predicate abstraction for programmable logic controllers. In: Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings, pp 123–138

Bjørner N, Gurfinkel A, McMillan KL, Rybalchenko A (2015) Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, pp 24–51

Blass A, Gurevich Y (1987) Existential fixed-point logic, Springer Berlin Heidelberg, pp 20–36

Bohlender D, Kowalewski S (2018a) Compositional verification of plc software using horn clauses and mode abstraction. IFAC-PapersOnLine 51(7):428 – 433, 14th IFAC Workshop on Discrete Event Systems WODES 2018

Bohlender D, Kowalewski S (2018b) Design and verification of restart-robust industrial control software. In: Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings, pp 47–68

Bohlender D, Simon H, Kowalewski S (2016) Symbolic verification of PLC safety-applications based on plcopen automata. In: 19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016., pp 33–45

Bohlender D, Hamm D, Kowalewski S (2018) Cycle-bounded model checking of PLC software via dynamic large-block encoding. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, pp 1891–1898

Carter M, He S, Whitaker J, Rakamaric Z, Emmi M (2016) SMACK software verification toolchain. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, pp 589–592

Cavada R, Cimatti A, Dorigatti M, Griggio A, Mariotti A, Micheli A, Mover S, Roveri M, Tonetta S (2014) The nuxmv symbolic model checker. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, pp 334–342

Clarke EM, Klieber W, Nováček M, Zuliani P (2012) Model Checking and the State Explosion Problem, Springer Berlin Heidelberg, pp 1–30

Darvas D, Fernandez Adiego B, Blanco E (2013) Transforming PLC Programs into Formal Models for Verification Purposes. Tech. rep., EN-ICE-PLC, CERN

Darvas D, Majzik I, Viñuela EB (2016) Formal verification of safety PLC based control software. In: Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings, pp 508–522

De Angelis E, Fioravanti F, Pettorossi A, Proietti M (2014) Verimap: A tool for verifying programs through transformations. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, pp 568–574

Dijkstra EW, Schölten CS (1990) The strongest postcondition, Springer New York, pp 209–215

Eén N, Mishchenko A, Brayton RK (2011) Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, pp 125–134

Frey G, Drath R, Schlich B, Eschbach R (2012) "safety automata" - A new specification language for the development of PLC safety applications. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation, ETFA 2012, Krakow, Poland, September 17-21, 2012, pp 1–8

Graf S, Saïdi H (1997) Construction of abstract state graphs with PVS. In: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings, pp 72–83

Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM pp 576–580

Hoder K, Bjørner N (2012) Generalized property directed reachability. In: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, pp 157–171

Komuravelli A, Bjørner N, Gurfinkel A, McMillan KL (2015) Compositional verification of procedural programs using horn clauses over integers and arrays. In: Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015., pp 89–96

Lange T, Neuhäußer MR, Noll T (2013) Speeding up the safety verification of programmable logic controller code. In: Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings, pp 44–60

Ljungkrantz O, Åkesson K, Fabian M, Yuan C (2010) Formal specification and verification of industrial control logic components. IEEE Trans Automation Science and Engineering 7(3):538–548

Manna Z, Pnueli A (1995) Temporal verification of reactive systems - safety. Springer

McMillan KL (1993) Symbolic model checking. Kluwer

Moon I (1994) Modeling programmable logic controllers for logic verification. IEEE Control Systems Magazine 14(2):53–59

de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pp 337–340

Ovatman T, Aral A, Polat D, Ünver AO (2016) An overview of model checking practices on verification of PLC software. Software and System Modeling 15(4):937–960

PLCopen TC5 (2006) Safety Software, Technical Specification, Part 1: Concepts and Function Blocks. PLCopen

PLCopen TC5 (2008) Safety Software, Technical Specification, Part 2: User Examples. PLCopen

Quinton S, Graf S (2008) Contract-based verification of hierarchical systems of components. In: Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008, pp 377–381

Simon H, Kowalewski S (2018) Mode-aware concolic testing for PLC software. In: Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings, pp 367–376