Design and Verification of Restart-Robust Industrial Control Software

Dimitri Bohlender(✉) and Stefan Kowalewski

Embedded Software, RWTH Aachen University, Aachen, Germany
{bohlender,kowalewski}@embedded.rwth-aachen.de

Abstract. Many systems in automated production and industrial automation operate in safety-critical environments and must meet rigorous safety requirements. To enable safe operation even in the case of a power outage, the PLCs driving these systems feature battery-backed memory areas to prevent loss of data and allow for implementation of resumption strategies. However it is up to an automation engineer to decide which variables to retain, and errors that only occur after program restart are a common problem in industrial control code.

We present approaches to both verifying the absence of such errors and synthesising safe configurations of retain variables with off-the-shelf tooling. The synthesis problem reduces to solving particular exists-forall quantified Horn clauses, for what we also propose a more efficient counterexample-guided procedure.

Evaluation of our prototypical implementation on examples from the PLCopen Safety library shows the techniques' strengths and limitations.

Keywords: Software verification · Parameter synthesis
Restart-robustness · Integration of formal methods
Programmable logic controllers

1 Introduction

In industrial applications, such as chemical plants or assembly lines, control software must meet high safety and reliability requirements as errors may entail significant costs and hazards. *Programmable logic controllers* (PLCs) are rugged computers which are particularly tailored to, and widely used in, the industrial automation domain.

The IEC 61131 standard defines requirements to both hardware and software aspects of PLCs as well as their cyclic mode of operation, i.e. reading inputs (from sensors), executing a main program, writing outputs (to actuators) and starting all over. To enable the design of systems that we call *restart-robust* w.r.t. some specification, i.e. whose behaviour complies with the specification even when resuming operation after a restart, the IEC 61131-3 defines a *retain* qualifier for variables in PLC programming languages. Variables that are declared this way, are stored in a dedicated battery-backed memory s.t. their values are available even after a power outage.

C. A. Furia and K. Winter (Eds.): IFM 2018, LNCS 11023, pp. 47–68, 2018.
https://doi.org/10.1007/978-3-319-98938-9_4

For example consider the use case of automated drilling of holes in workpieces. If the drill's position or mode of operation are not retained, a restart may result in unintended movement of the drill and damage to the system itself, the payload, or persons within reach – even if such a malfunction were not possible in a restart-free operation.

While retain variables are standardised, the semantics of an assignment to such a variable is not, and left to the PLC vendors. In this work, we focus on the two most prominent implementations supported by major development environments for PLC software: *immediate* and *delayed* writing of retain variables. In the former case, an assignment to a retain variable is translated to an immediate write to the battery-backed memory. However, frequent writing to this memory is often slower than accessing the main memory, and every immediate write increases the number of corner-cases to consider when developing restart-robust applications. By way of contrast, in the case of delayed writing, assignments to retain variables during program execution are in fact writes to the main memory. The actual copying of these values to the battery-backed memory is delayed until the end of the current PLC cycle. Depending on the application or PLC vendor, one or the other semantics may have to be supported.

Due to the cyclic operation of PLCs, where outputs are only written at the end of a program execution, the intermediate states of a PLC are not visible to the environment. Therefore, when automation engineers or specifications talk about a PLC's state they implicitly refer to its *observable state*. Although most specifications in this domain are formulated in natural language, they can usually be expressed formally in terms of invariants or temporal logics [16]. When developing restart-robust control software or upgrading existing functionality to handle restarts safely, it becomes an automation engineer's task to manually determine which variables must be retained without violating a given specification of safe behaviour, and implement the functionality needed for resuming operation. Since mistakes can easily be made, but be very subtle and hard to detect, this is a common problem in industrial control code [20].

Contribution. The primary contribution of this paper is the design of automated verification procedures that aid in the engineering of restart-robust logic control software. To this end,

1. we formalise the restart behaviour for delayed and immediate write semantics, and sketch its integration with established approaches for PLC software verification,
2. we show how these characterisations can be extended, to acquire procedures that synthesise restart-robust configurations of retain variables using off-the-shelf tools,
3. we propose a dedicated counterexample-guided procedure, which exploits specifics of the problem, and makes synthesis of configurations practical in the first place,

4. we evaluate our approaches to both verification of a program's restart-robustness and synthesis of safe configurations of retain variables using examples from the *PLCopen Safety* [31,32] library, and
5. we provide all the artefacts needed to reproduce our results, or even improve upon.

Related Work. Due to the safety critical nature of industrial automation, the use of formal verification is advisable 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 [30]. 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.

The endeavour of verifying PLC software goes back to Moon [28], who used the SMV formalism [27] to characterise programs written in the *Ladder Diagram* programming language. 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 SMV-based tooling for model checking higher-level PLC programming languages [3,16,30]. However, with *constrained Horn clauses* (CHCs) increasingly becoming a basis for automatic program verification in recent years [7], they have been adopted in verification of logic control software too [9,10]. Therefore, we examine the characterisation of restart semantics in both formalisms.

To the best of our knowledge, we are the first to investigate formal verification of a program's restart-robustness and synthesis of safe retain configurations. The only directly related work [20] assumes delayed write semantics and adapts static value analysis to distinguish between variables' values before and after a restart. *Crash recoverability* of C programs [24] is a related problem, using a similar modelling, but differing from restart-robustness in terms of requirements and program transformations.

The search for constants s.t. a system satisfies some property is commonly referred to as *parameter synthesis*, and we model the search for safe retain configurations as such. Besides of our characterisation of the problem in terms of the SMV formalism, SMV-based tooling has also been used in bioinformatics to find parameters for models of gene regulatory networks [2]. Our counterexample-guided approach is most similar to [13] but does not require quantifier elimination, is independent of the chosen theory to model values, and works with any CHC-solving algorithm.

Outline. We commence with an example program, illustrating the concrete problems and expected solutions. Section 3 recapitulates the formal concepts relevant for characterisation of these problems in terms of existing formalisms (Sect. 4) and understanding of the counterexample-guided synthesis procedure (Sect. 5). In Sect. 6, we present experimental results and provide concluding remarks in Sect. 7.

2 Motivating Example

Consider the program from Fig. 1, which picks up on the example used in [20], but is slightly modified to make a different point. For simplicity, it does not feature input variables and operates on two integer variables `a` and `b`, and a retentive Boolean flag `fs`. Intuitively, the flag `fs` is used to track whether the program is in its first cycle, s.t. the initialisation of `b` (cf. line 11) is only executed once. The program starts with the explicitly provided default initialisation $[fs \mapsto true, a \mapsto 0, b \mapsto 0]$.

```
1  PROGRAM Program1
2    VAR RETAIN
3      fs:BOOL := TRUE;
4    END_VAR
5    VAR
6      a:INT := 0;
7      b:INT := 0;
8    END_VAR
9    IF fs THEN
10     fs := FALSE;
11     b := 2;
12   END_IF
13   a := 1234/b;
14 END_PROGRAM
```

Fig. 1. Running example program

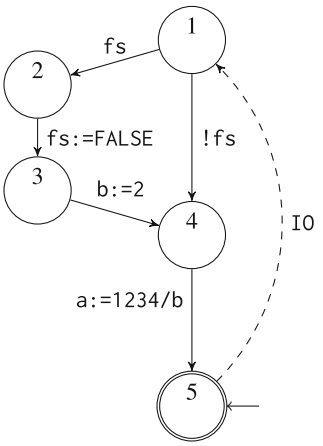

Fig. 2. CFA of the running example

Let $a \geq 0$ be the invariant that needs to hold for every observable state. In a regular execution this is indeed true. Both `a` and `b` are initially set to 0, but `b` is set to 2 during the first cycle which results in `a` being set to $1234/2 = 617$. Since `fs` is only *true* in the first cycle, the values of `b` and `a` stay like that forever.

However, the program is not restart-robust w.r.t. to $a \geq 0$ in the context of delayed write semantics for retain variables. If a restart occurs after the first cycle, i.e. once $\neg fs \wedge b = 2$ holds, `fs` will stay unchanged but `b` will be reset to 0 and let `a` take an arbitrary value, by causing an undefined division by 0. Since immediate write semantics allow for a superset of the behaviours of delayed write semantics, the program is not restart-robust for those either.

The next question is, whether it is possible to fix the program by changing which variables to retain. The variable `fs` that is being retained currently, is assumed to be retained for a reason and not in question to become volatile, but close inspection and intuition will help with identifying `b` as a suitable candidate. If `b` is retained too, the program becomes restart-robust w.r.t. $a \geq 0$, since the divisor in line 13 is always 2.

Nevertheless, if the program is used in the context of immediate write semantics for the retain variables, it suddenly becomes not restart-robust again. Although the program is simple, the violating run is easy to miss and will eventually lead to unexpected behaviour. With immediate write semantics, a restart might occur after the write to `fs` but before setting `b` to 2, i.e. leaving it at its initial 0. Since $\neg fs$ will hold, there will be no initialisation and the division by 0 will be reached again. In fact, in this case there is no way to achieve restart-robustness by changing the configuration of retain variables.

Keeping track of all the possible (mis-)behaviour in the context of restarts is clearly prone to human error. It should not be surprising that unexpected behaviour after a restart is a common problem [20], given that implementation of restart-robust systems is currently approached without aid of automatic procedures.

3 Preliminaries

3.1 Program Representation

We restrict the presentation to a reduced programming language, featuring only *assignments*, *assumes* and a *havoc* instruction, which models the assignment of a nondeterministic value, visualised as $x := e$, g and $x :=?$ respectively, where x is a variable, e an expression, and g a Boolean expression acting as a guard. This is a common approach [1,7,8], coming without loss of generality. In particular, all calls can be inlined since recursion is prohibited in PLC programs.

We use *Instr* to denote the set of such instructions, and represent a program using the notion of a *control flow automaton* (CFA).

Definition 1 (Control Flow Automaton). *A CFA $\mathcal{A} = (L, E)$ is a directed graph, where the vertices L are the program locations, and the edges $E \subseteq L \times Instr \times L$ model the program's instructions and their effect on control flow.*

Definition 2 (Program). *A program $P = (\mathbf{X}, \mathbf{X}_{in}, \mathcal{A}, l_{EoC}, l_{init}, def)$ consists of a set of variables $\mathbf{X}$, input variables $\mathbf{X}_{in} \subseteq \mathbf{X}$, a CFA $\mathcal{A}$ whose instructions refer to the variables from $\mathbf{X}$, the end-of-cycle location $l_{EoC} \in L$, the initial program location $l_{init} \in L$, and a partial mapping def from variables to their default values. The characterisation $I(\mathbf{X})$ of initial values is implicitly given by the defined defaults, i.e. $\bigwedge_{x \in \mathbf{X}} x = def(x)$.*

Note that l_{EoC} is both the only location where a PLC's state is observable and also the initial program location, although the formalism allows for them to differ, e.g. as a result of instrumentation (cf. Sect. 4.1). Similarly, while all variables of a logic control application have well-defined default values (cf. IEC 61131), later modelling steps may introduce variables for which *def* is undefined and the initial value nondeterministic.

Figure 2 illustrates the CFA that models our running example. For the sake of readability and intuition, we use an `IO` instruction to model the PLC's reading

of sensor values at the beginning of a new cycle, which is syntactic sugar for a sequence of $x :=?$ for each $x \in \mathbf{X}_{\text{in}}$. Since this particular program has no input variables anyway, it can be thought of as a plain *goto*, representable as the assume $(5, \mathit{true}, 1) \in E$.

Semantics. The *state* of a program is an assignment σ that maps each variable from $\{pc\} \cup \mathbf{X}$ to a value, where pc represents the program counter, i.e. $\sigma(pc) \in L$.

Since a CFA is essentially a GOTO-program, its transition relation $T \subseteq \Sigma \times \Sigma$, where Σ denotes the set of all states, can be derived from the *weakest-preconditions* of unstructured programs [1], i.e.

$$T(pc, \mathbf{X}, pc', \mathbf{X}') = \bigwedge_{(l, \mathit{instr}, l') \in E} (pc = l \rightarrow [\![\mathit{instr}]\!] \wedge pc' = l'), \tag{1}$$

where the primed variables' instances $\{pc'\} \cup \mathbf{X}'$ denote the next-state valuation and $[\![\cdot]\!]$ is the instruction's characterisation:

$$[\![\mathit{instr}]\!] = \begin{cases} (\bigwedge_{v \in \mathbf{X} \setminus \{x\}} v' = v) \wedge x' = e & \mathit{instr} = (x := e) \\ (\bigwedge_{v \in \mathbf{X} \setminus \{x\}} v' = v) & \mathit{instr} = (x :=?) \\ (\bigwedge_{v \in \mathbf{X}} v' = v) \wedge g & \mathit{instr} = g \end{cases} \tag{2}$$

3.2 Symbolic Model Verifier

The *Symbolic Model Verifier* (SMV) formalism allows the symbolic definition of a transition system $S = (\mathbf{V}, I, T)$ in terms of a characterisation of the initial states I over the variables $\mathbf{V}$, and a transition relation T as seen in the previous section. Accordingly, modelling the program semantics for SMV-based verifiers is a straight-forward reuse of Eq. (1). Note though that SMV targets hardware-verification, and with $\mathbf{V} = \{pc\} \cup \mathbf{X}$ the program counter is treated like any other variable. Therefore, if the control flow is to be exploited by a SMV-based verifier, some variant of *large-block encoding* [4] has to be employed.

The SMV formalism allows the definition of specifications in terms of *invariants* and *temporal logics* – in particular CTL [14]. However, one needs to exercise caution when expressing a specification for PLC software in SMV, since SMV specifications are interpreted in the step-size that T is provided in, e.g. a single instruction per step, while the original specification only refers to the observable states, i.e. a step is a whole execution cycle. For an invariant $\varphi(\mathbf{X})$ this can be easily accounted for by only checking it at the end-of-cycle location, i.e. use $pc = l_{\text{EoC}} \rightarrow \varphi(\mathbf{X})$. Although we focus on invariants, the need for reformulation of specifications can generally be avoided by characterising the whole program as a single step [3,8].

3.3 Constrained Horn Clauses in Software Verification

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

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

where φ is a constraint over $\mathcal{F}$ and $\mathcal{V}$, $\mathbf{X}_i, \mathbf{X} \subseteq \mathcal{V}$ 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 $\mathcal{P}$ symbols and otherwise, it is called a *rule*. Following the convention of logic programming literature, we use the shorthand notation

$$h(\mathbf{X}) \leftarrow p_1(\mathbf{X}_1), \ldots, p_k(\mathbf{X}_k), \varphi. \tag{3}$$

A set of CHCs is satisfiable if there exists an interpretation of the predicates that satisfies each φ. As illustrated by [22], intuitively, each p_i represents an unknown over-approximate summary, while a query defines a property to be proved. In the context of CFAs, the p_i correspond to over-approximations of the reachable valuations at program location i. Therefore, checking whether a program satisfies a safety property, amounts to establishing the satisfiability of CHCs that encode the corresponding verification conditions, as shown below.

Following [7], a program $P = (\mathbf{X}, \mathbf{X}_{\text{in}}, (L, E), l_{\text{EoC}}, l_{\text{init}}, def)$ is characterised by

$$p_{\text{init}}(\mathbf{X}) \leftarrow I(\mathbf{X}) \tag{4}$$

$$p_{l'}(\mathbf{X}') \leftarrow p_l(\mathbf{X}), [\![instr]\!] \text{ for each } (l, instr, l') \in E \tag{5}$$

Note that in contrast to the p_i, I is not uninterpreted but explicitly given (cf. Sect. 3.1).

To prove that the program complies with an invariant φ, we check whether an interpretation of predicates p_i exists s.t. all CHCs are satisfied and the over-approximation of observable states $p_{\text{EoC}}(\mathbf{X})$ subsumes the safe states (cf. [26]), by adding the query

$$\varphi(\mathbf{X}) \leftarrow p_{\text{EoC}}(\mathbf{X})\,. \tag{6}$$

4 Modelling the Restart Semantics

Existing approaches for PLC software verification formalise only the *nominal* program semantics, implementing the approaches from Sect. 3, and ignoring possible restarts. In the following, we illustrate how restarts with delayed and immediate write semantics for retain variables can be modelled in terms of these established formalisms, to allow reuse of existing verification machinery.

Similar to an interrupt, a restart may occur at any time during program execution. When that happens, the program counter is reset to l_{EoC}, and the next execution cycle starts with all non-retain variables reinitialised with their default values. The retain variables, however, take their corresponding values that are stored in the battery-backed memory at that time. Note that marking a variable as retained does not imply that all assignments to it are immediately reflected in the battery-backed memory – this depends on the employed retain semantics.

Keep in mind that for PLC programs, the initial and end-of-cycle location are identical, and the following instrumentations are to be employed prior to other modifications that may introduce a distinct entry l_{init} for modelling purposes (cf. Definition 7).

Delayed Write Semantics. If a logic controller is restarted in the middle of an execution cycle and writing to retain variables is realised via delayed write semantics, there will not have been any write to the battery-backed memory since the end of the previous cycle. The resulting state will have all non-retain variables reset to their initial values, and the retain variables back at the values they had at the end-of-cycle location.

Since a cycle's nominal semantics becomes irrelevant if a restart happens during its execution, we model such a restart by a nondeterministic choice at the end-of-cycle location. If a restart is chosen to occur, we can keep the current values of all retain variables and reinitialise the others, otherwise we execute the program's nominal semantics.

Note that power outages during the delayed writes are omitted in the modelling since these writes can be handled atomically by the PLC's operating system, e.g. by using auxiliary memory-backed variables that are written immediately.

Definition 4 (Delayed Write Instrumentation). *Given a set of retain variables $\mathbf{X}_{ret} \subseteq \mathbf{X}$ for a program $P = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_{EoC}, l_{EoC}, def)$, its* delayed write instrumentation *yields a program*

$$P_{dw} = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, def)$$

where

- $L_{init} := \{l_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new program locations in between which the resetting of values occurs – one for each non-retain variable,*
- $E_{init} := \{(l_{x_1}, x_1 := def(x_1), l_{x_2}), \ldots, (l_{x_n}, x_n := def(x_n), l_{EoC})\}$ *are the reinitialising assignments for every non-retain variable,*
- $E_{restart} := \{(l_{EoC}, true, l_{x_1})\}$ *models a restart during the execution of the cycle,*

with the non-retain variables denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$.

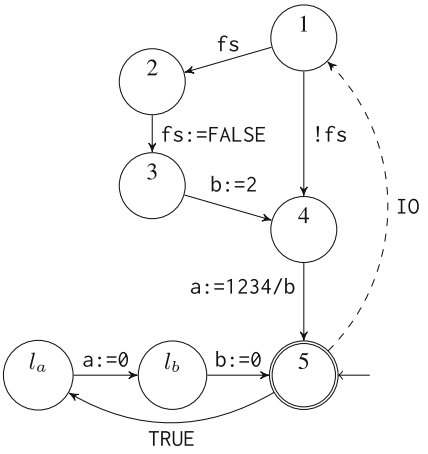

Fig. 3. Delayed write instrumentation

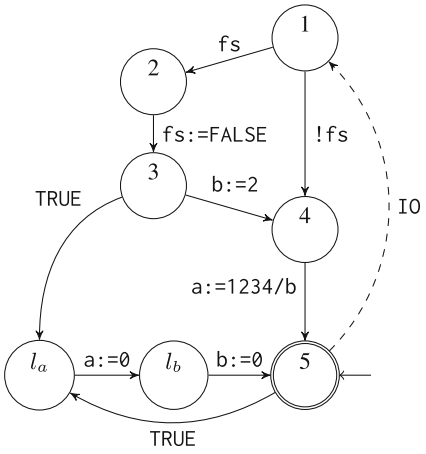

Fig. 4. Immediate write instrumentation

Figure 3 illustrates the result of applying this instrumentation to our running example. At the beginning of every execution cycle, either the edge $(5, \mathit{true}, l_a)$ leading to the reinitialisation, or the `IO`-edge leading to the nominal cycle semantics will be taken.

Immediate Write Semantics. If a logic controller is restarted in the middle of an execution cycle and writing to retain variables is realised via immediate write semantics, all the assignments to retain variables on the path from the end-of-cycle location to the location where the restart occurred will be reflected in the battery-backed memory. The resulting state will have all non-retain variables reset to their initial values, and the retain variables at the values they had at the time of the restart.

Since only assignments to retain variables change the resulting state after a restart, it suffices to model restarts with a nondeterministic choice after every write to a retain variable – instead of a choice in every location. Note that, although a restart that occurs before any write to a retain variable does not lead to a new state, and is irrelevant for checking invariants, it may still cause violations of other temporal specifications. To model the restarting before any write to a retain variable, we also add a nondeterministic choice at the end-of-cycle location, as in the case of delayed write semantics.

Definition 5 (Immediate Write Instrumentation). *Given a set of retain variables* $\mathbf{X}_{ret} \subseteq \mathbf{X}$ *for a program* $P = (\mathbf{X}, \mathbf{X}_{in}, (L, E), l_{EoC}, l_{EoC}, \mathit{def})$*, its* immediate write instrumentation *yields a program*

$$P_{iw} = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, \mathit{def})$$

where

- $L_{init} := \{l_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new program locations in between which the resetting of values occurs – one for each non-retain variable,*
- $E_{init} := \{(l_{x_1}, x_1 := def(x_1), l_{x_2}), \ldots, (l_{x_n}, x_n := def(x_n), l_{EoC})\}$ *are the reinitialising assignments for every non-retain variable,*
- $E_{restart} := \{(l_{EoC}, true, l_{x_1})\} \cup \{(l', true, l_{x_1}) \mid (l, x := e, l') \in E, x \in \mathbf{X}_{ret}\}$ *models a restart before any and after every write to a retain variable during the cycle,*

with the non-retain variables denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$.

Note that with L_{init} and E_{init} being the same as in the delayed write instrumentation, but more cases of restarts to consider, the immediate write instrumentation yields a superset of its behaviours. As a result, if a program is restart-robust in the context of these semantics, it is also restart-robust in the context of delayed write semantics. If a program is not restart-robust in the context of delayed write semantics, it will not be in the context of these semantics either.

Figure 4 illustrates the result of applying this instrumentation to our running example. Besides the restart edge at the end-of-cycle location, which models a restart occurring prior to any write to the battery-backed memory, we now also consider a restart after the write to *fs*, since it was declared as a retain variable.

To check whether a program is restart-robust w.r.t. to some specification, we can now use the appropriate instrumentation to reduce the problem to something, that we already have verification procedures for (cf. Sect. 3).

4.1 Characterising Parameter Synthesis with CHCs and SMV

While the proposed reductions enable checking a program's restart-robustness w.r.t. some specification, they do not aid the developing engineer in actually designing programs that are restart-robust, or upgrading existing modules to enable restarts-robustness by choosing appropriate retain variables. Therefore, this section examines how the presented reductions can be modified, s.t. existing tooling can also be used to synthesise configurations of retain variables that make the program restart-robust w.r.t. a property of interest.

To enable the examination of different configurations of retain variables, the configuration itself must become a *parameter* of the model. To this end, we add Boolean constants to the model, one for each non-retain variable, which encode whether the corresponding variable is to be retained. The constants' values are nondeterministically chosen at the start of the program and used to parametrise the reinitialisation semantics. Furthermore, they are used to guard the restarts that depend on whether a particular variable is retained, e.g. in the case of immediate write semantics.

Definition 6 (Parametrisation of Retains). *Given the result of a delayed or immediate write instrumentation* $P = (\mathbf{X}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{init} \uplus E_{restart}), l_{EoC}, l_{EoC}, def)$ *and the used retain variables* $\mathbf{X}_{ret} \subseteq \mathbf{X}$*, its* parametrisation of retains *yields a program*

$$P_{par} = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in}, (L \uplus L_{init}, E \uplus E_{parInit} \uplus E_{restart} \uplus E_{parRestart}), l_{EoC}, l_{EoC}, def)$$

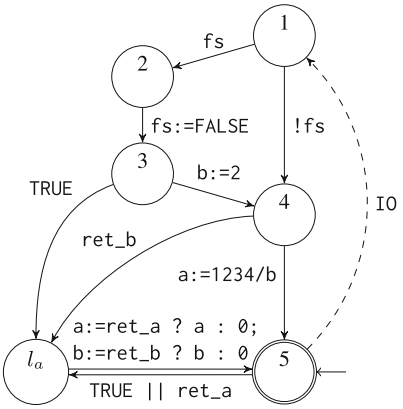

Fig. 5. Immediate write instrumentation with dependence on retain configuration

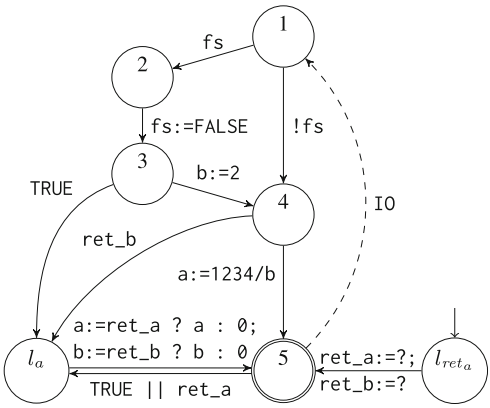

Fig. 6. SMV-based synthesis requires choice of retain variables to be part of the model

where the non-retain variables are still denoted by $x_1, \ldots, x_n = \mathbf{X} \setminus \mathbf{X}_{ret}$*, and*

- $\mathbf{X}_{par} := \{ret_x \mid x \in \mathbf{X} \setminus \mathbf{X}_{ret}\}$ *are new Boolean variables that parametrise which of the currently non-retained variables to treat as retained,*
- $E_{parInit} := \left\{ \begin{array}{c} (l_{x_1}, x_1 := ret_{x_1} ? x_1 : def(x_1), l_{x_2}), \ldots, \\ (l_{x_n}, x_n := ret_{x_n} ? x_n : def(x_n), l_{EoC}) \end{array} \right\}$
  are the parametrised reinitialising assignments, using ternary if expressions,
- $E_{parRestart} := \begin{cases} \emptyset & \text{delayed write} \\ \{(l', ret_x, l_{x_1}) \mid (l, x := e, l') \in E, x \in \mathbf{X} \setminus \mathbf{X}_{ret}\} & \text{otherwise} \end{cases}$
  models a restart after every write to a variable that can be parametrised to be retained in the case of immediate write semantics.

Figure 5 illustrates the result of parametrising the retain configuration, applied to an immediate write instrumentation of our example program. In comparison to Fig. 4, every static reinitialisation of a non-retain variable x has been replaced with an expression dependent on a parameter ret_x, and additional guarded edges that lead to the resetting have been added after assignments to potential retain variables. Note that for the sake of readability, the visualisation of the reinitialisation sequence is simplified in that both assignments are presented as a sequence on a single edge, instead of featuring the intermediate location l_b as in previous figures.

However, analysing a parametrised program with the techniques from Sect. 3 will not result in checking whether a restart-robust retain configuration exists, but whether all possible retain configurations (of not yet retained variables) are restart-robust.

Existential Quantification. To determine whether a retain configuration exists that makes the program restart-robust w.r.t. a property of interest, we illustrate how the introduced parameters can be existentially quantified in the context of both the CHC and SMV formalism.

Since the variables in CHCs are implicitly universally quantified, the synthesis problem requires us to move on to the more complex case of *exist-forall* quantified CHCs. Expressing parameter synthesis in this class of Horn clauses is straightforward. We keep the actual clauses as illustrated in Sect. 3.3, but replace the quantification $\forall \mathcal{V}$ by $\exists \mathbf{X}_{\text{par}} \forall \mathcal{V} \setminus \mathbf{X}_{\text{par}}$. The resulting constrains are satisfiable if interpretations for both the parameters $x \in \mathbf{X}_{\text{par}}$ and utilised predicates $p \in \mathcal{P}$ exist, s.t. the clauses are satisfied for all values of the remaining variables $\mathcal{V} \setminus \mathbf{X}_{\text{par}}$. The downside is that a solver will not be able to use its efficient procedures tailored to solving universally quantified Horn clauses, but resort to general techniques for *satisfiability modulo theories* (SMT) [25].

While the SMV formalism itself does not support quantification, a CTL specification may reason about the existence of a path. In combination with another modification of the CFA, the existence of a certain retain configuration can be reduced to the existence of a path. Intuitively, we prepend a nondeterministic choice of a retain configuration to the original program entry l_{EoC}, and question the existence of a path through this choice s.t. from l_{EoC} on the program exhibits restart-robust behaviour.

Definition 7 (Integration of Parameter Choice). *Given a delayed or immediate write instrumented and parametrised program* $P = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in},)$ $(L, E), l_{EoC}, l_{EoC}, def$*, the* integration of parameter choice *yields a program*

$$P_c = (\mathbf{X} \uplus \mathbf{X}_{par}, \mathbf{X}_{in}, (L \uplus L_c, E \uplus E_c), l_{EoC}, l_{x_1}, def)$$

where the parameters are denoted by $x_1, \ldots, x_n = \mathbf{X}_{par}$*, and*

- $L_c := \{l_x \mid x \in \mathbf{X}_{par}\}$ *are new program locations in between which the choice of retain variables occurs – one for each parameter,*
- $E_c := \{(l_{x_1}, x_1 :=?, l_{x_2}), \ldots, (l_{x_n}, x_n :=?, l_{EoC})\}$ *are the actual nondeterministic choices for every parameter.*

Figure 6 illustrates the result of integrating parameter choice into our running example, assuming immediate write semantics. Note that, as in Fig. 5, we use sequences of assignments instead of putting them on separate edges to avoid clutter. This time, similar to the reinitialising assignments, we prepend a sequence of havoc-instructions that realise the nondeterministic choice of a retain configuration, before the actual program semantics are considered. To check whether a retain configuration exists s.t. the program is restart-robust w.r.t. the invariant $\varphi(\mathbf{X})$, it suffices to formalise the program in the usual way (cf. Sect. 3.2) and check whether the CTL specification $\mathsf{EX}\ \mathsf{EX}(pc = l_{\text{EoC}} \rightarrow \varphi(\mathbf{X}))$ holds.

With the illustrated approach, the CTL formula will always need as many EX as parameters are present, to quantify over the prepended path up to l_{EoC}. Note that in practice, this sequence of choices will usually be characterised as a

single composite choice, and a single EX will suffice. Unfortunately, as with the CHC-based modelling, switching to CTL will result in more general procedures being used by a verifier.

5 Counterexample-Guided Synthesis of Safe Retain Configurations

Due to the need for existential quantification in parameter synthesis, a reduction to the previous formalisms will result in significantly more complex decision procedures being used. However, our use of existential quantification is very specific in that we only quantify over Boolean variables and their values also stay constant throughout the possible executions. Therefore it seems natural to manage the choice of parameters oneself, and reuse the efficient procedures for reasoning about restart-robustness for fixed parameters.

Counterexample-guided abstraction refinement (CEGAR) [15] is a general framework for computing an over-approximation, by finding counterexamples that reveal issues with the current approximation and improving it w.r.t. them iteratively. Similar to [13], we use this scheme to over-approximate the supposedly "safe" choices for parameters and refine them iteratively, until all that remains is a characterisation of choices that are guaranteed to exhibit only restart-robust behaviour.

In Sect. 4.1 we have seen that the universally quantified CHCs of our parametrised program check whether all parameter choices lead to restart-robust behaviour. If we had a guess at a characterisation $\mathit{safe}(\mathbf{X}_{\text{par}})$ of safe choices, the same machinery could be used to prove that all these choices indeed result in restart-robustness, by checking whether our CHCs are satisfiable in the context of the following query

$$\varphi(\mathbf{X}) \leftarrow p_{\text{EoC}}(\mathbf{X} \uplus \mathbf{X}_{\text{par}}), \mathit{safe}(\mathbf{X}_{\text{par}}). \tag{7}$$

If no satisfying interpretation of predicates exists, *safe* is a wrong guess and the CHC solver will provide a counterexample that describes a run through the CFA to an end-of-cycle location where φ is violated, and in particular yield the concrete Boolean parameters that led to this. For the next iteration, one would improve *safe* by excluding the apparently bad choice from it.

Algorithm 1 shows the pseudocode of our procedure that follows this intuition. To begin with, the already instrumented and parametrised program P is characterised in terms of both universally quantified CHCs and a symbolic transition system, as presented in Sects. 3.3 and 3.2 respectively. Remember that in the symbolic transition system the program counter is part of the variables, and its $\mathbf{V} = \{pc\} \cup \mathbf{X} \uplus \mathbf{X}_{\text{par}}$ should not to be mistaken for the $\mathcal{V} = \mathbf{X} \uplus \mathbf{X}_{\text{par}}$ used in the CHCs.

The main loop, starting in line 4, implements the refinement procedure described above. If no counterexample is found, *safe* already characterises the retain configurations that lead to restart-robust behaviour. Note that the

Algorithm 1. SynthRetainConf(P, φ)

Input : Program $P = (\mathbf{X} \uplus \mathbf{X}_{\text{par}}, \mathbf{X}_{\text{in}}, \mathcal{A}, l_{\text{EoC}}, l_{\text{EoC}}, def)$ with prametrised retains
Predicate $\varphi(\mathbf{X})$ characterising safe states
Variables: Predicate $safe(\mathbf{X}_{\text{par}})$ charactering parameters that do not lead to violations
Universally quantified Horn clauses $\mathcal{H}$

1 $\mathcal{H} \leftarrow \mathsf{toHorn}(P)$ // Represent program as ∀CHCs
2 $(\mathbf{V}, I, T) \leftarrow \mathsf{toSymTS}(P)$ // and as symbolic transition system
3 $safe(\mathbf{X}_{\text{par}}) \leftarrow true$ // All parameters are assumed to be safe
4 **while** ¬sat $(\mathcal{H} \cup \{\varphi(\mathbf{X}) \leftarrow p_{EoC}(\mathbf{X} \uplus \mathbf{X}_{par}), safe(\mathbf{X}_{par})\})$ **do** // ∃ violating run?
5 $k \leftarrow$ length of violating run
6 $c_{\text{par}} \leftarrow$ cube of chosen (Boolean) parameter values in violating run
7 **foreach** *lit in* c_{par} **do**
8 $\bar{c}_{\text{par}} \leftarrow c_{\text{par}}$ with negated *lit* // Flip literal
9 **if** sat $(I(\mathbf{V}) \wedge \bigwedge_{0 \le i < k} T(\mathbf{V}_i, \mathbf{V}_{i+1}) \wedge \bar{c}_{par} \wedge \neg\varphi(\mathbf{X}_k))$ **then** // Still violating?
10 $c_{\text{par}} \leftarrow c_{\text{par}} \setminus lit$ // Drop literal
11 $safe(\mathbf{X}_{\text{par}}) \leftarrow safe(\mathbf{X}_{\text{par}}) \wedge \neg c_{\text{par}}$ // Block unsafe parameters
12 **return** $safe(\mathbf{X}_{\text{par}})$ // (Potentially empty) region of safe parameters

returned predicate will characterise an empty set if no such configuration exists. However if a counterexample exists, we determine its length and the chosen retain configuration, as a conjunction of literals, to prevent the same choice being taken in future iterations (cf. line 11).

To avoid enumerating every single counterexample, CEGAR schemes usually generalise the found counterexample, s.t. a set of counterexamples that violate the specification for the same reason can be excluded at once. Unlike in the general setting of parameter synthesis [13], we do not need special quantifier elimination procedures for the theories that the other variables are represented in, but can adapt generalisation strategies for Boolean cubes.

Similar to the directed but expensive approach of explicitly trying to remove the literals one by one [17], the loop starting in line 7 iterates over every literal once and probes whether it affects the reachability of a violation. To this end, knowing the length of the counterexample, we construct a *bounded model checking* (BMC) [6] instance that characterises the possible executions up to a violation at this length (cf. line 9). To allow for the variables' values to change between different steps, the BMC query uses several instances $\mathbf{V}_i$ of the variables, and $\mathbf{V}_0$ simply denotes $\mathbf{V}$.

Every literal from the cube c_{par}, that characterises the unsafe choice, is then iteratively flipped to determine its impact on the violation. If flipping a literal still leads to a violation, the literal is irrelevant and removed from the cube during iteration (cf. line 10). Note that we use set operations on cubes, like set difference or the subset relation, to denote the operations on the cubes' sets of literals.

Theorem 1 (Generalisation is Sound). *The proposed generalisation procedure always yields a cube $c_g \subseteq c_{par}$ that characterises only unsafe choices.*

Proof. Since the procedure only removes literals from c_{par} to acquire the resulting cube, the relation $c_g \subseteq c_{\text{par}}$ holds by construction. It remains to prove that c_g contains unsafe choices only, i.e. for all choices characterised by c_g, a violating run of length k exists:

$$\forall_{\mathbf{X}_{\text{par}}} c_g(\mathbf{X}_{\text{par}}) \rightarrow \exists_{\mathbf{V} \setminus \mathbf{X}_{\text{par}}, \mathbf{V}_1, \ldots, \mathbf{V}_k} I(\mathbf{V}) \wedge \bigwedge_{0 \leq i < k} T(\mathbf{V}_i, \mathbf{V}_{i+1}) \wedge \neg\varphi(\mathbf{X}_k). \quad (8)$$

Base case: The c_{par} that the generalisation is entered with, characterises a single choice that can lead to a violation, so for $c_g = c_{\text{par}}$ formula (8) holds trivially.

Inductive step: Let the formula hold for some c_g. Flipping a literal *lit* in c_g yields $\bar{c}_g$, and two outcomes for the BMC query with $\bar{c}_g$ have to be considered:

- If the query is satisfiable, a violation is reachable even with $\neg lit$ instead of *lit*. Since both c_g and $\bar{c}_g$ apparently characterise unsafe choices, the formula still holds for their disjunction $c_g \vee \bar{c}_g$, which simplifies to the $c_g \setminus lit$ that we keep.
- If the query is unsatisfiable, $\bar{c}_g$ is a safe configuration and c_g, for which the formula is known to hold, will not be modified.

In fact, this approach is an *anytime algorithm*, since no matter in which order the literals are probed, the formula always stays valid and generalisation can be stopped at any time.

While a single iteration over all literals is not guaranteed to yield the most general form, it already has a significant impact and is cheaper than repeating the procedure until a fixed-point is reached.

6 Experiments

Implementation Details. We implemented Java-prototypes of both the reduction-based and counterexample-guided approach, using the publicly available SMT solver Z3 [29] and the ARCADE.PLC platform for analysis of PLC software [5]. Unlike the presented characterisation of single instructions, we implement the common approach of encoding the whole execution cycle as one step [3,8], which is required for efficient reasoning [4].

We use nuXmv [12] and Z3 as off-the-shelf verifiers for the SMV and CHC formalisms that we reduce the verification tasks to. Although the analysed programs do not feature complex operations on bitvectors, and could as well have been modelled with unbounded integers, we characterise the semantics through the theory of fixed-size bitvectors since nuXmv does not support CTL checking over infinite domains, and to the best of our knowledge, no other SMV-based verifier does either.

Due to a bug in the latest version (4.6.0) of Z3 which causes segmentation-faults on retrieval of certain counterexamples, the prototype of our guided approach is linked with an older version (4.5.0) which does not feature CHC-solving with SPACER [23] yet but uses the usually slower *Property Directed Reachability*

(PDR) [21] instead. Since our approach is agnostic about the employed CHC solving procedure, the switch amounted to changing a single parameter.

Furthermore, we do not construct the BMC instance in Algorithm 1 anew in every iteration, but reuse the same one in an incremental fashion and realise the probing for violations by *solving under assumptions* [18].

Benchmarks. The PLCopen is an organisation which drives standardisation and technical specifications in automation. The *PLCopen Safety* library is a collection of such specified modules for domain-specific problems, e.g. how to realise a safe emergency shutdown. We experimented with two groups of PLC programs from this library, whose sizes range from 117 to 1450 program locations per cycle. Programs from [31] are elementary modules, each one implementing a particular safety concept, while [32] features user examples which combine these to form more complex applications.

The considered specifications are invariants that have been used in other case studies [8] and were either formulated by the PLCopen or derived from their technical specifications – the concept is applicable to all specifications that can be reduced to reachability checking though. The benchmark encompasses 56 specifications, 37 of which concern the elementary modules, while the remaining 19 refer to the composite applications.

These programs were not designed with restarts in mind, so we investigate whether they happen to nevertheless be restart-robust w.r.t. the specifications, and whether safe retain configurations exist at all. Since the elementary modules exhibit state-machine semantics, featuring a `DiagCode` variable that tracks the current mode of operation, we declare it to be a retain variable beforehand – similar to `fs` in our running example.

Since the encoding of an execution cycle as a single step is negligibly fast and needs to be performed only once for each program, independent of the checked specification or chosen backend-verifier, we only compare the CPU time spent by the verifiers to allow for a direct comparison of the techniques.

All experiments were performed on a 64 bit Linux machine with 3.5 GHz, 8 GB of RAM and a timeout of 1800 s. They can be reproduced with the artefacts available on our website[1]. Note that for clarity, this package also features analysis results of the running example, and the actual CFAs and their encodings.

Results of Restart-Robustness Checking. In the following, we discuss the measurements for our experiments on verifying restart-robustness w.r.t. a given specification, using the formalisation presented in Sect. 4. To this end, we measure the time it takes to check a specification on the original program, treating retain variables like regular ones, and the time spent on delayed and immediate write instrumented variants of the program. Overall this results in 168 verification tasks.

[1] https://arcade.embedded.rwth-aachen.de/ifm18_restart.tar.gz.

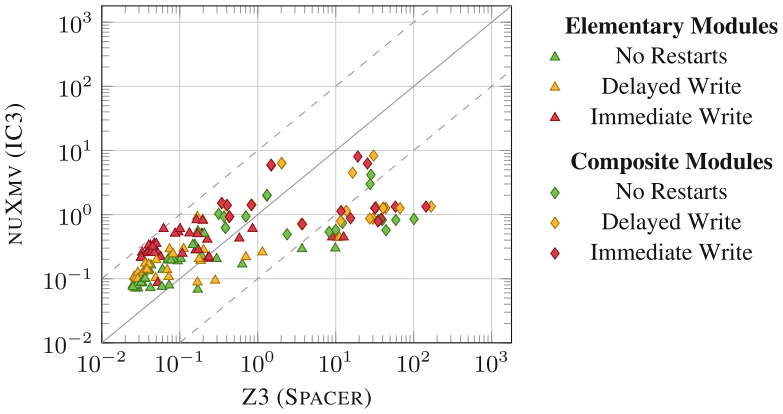

Fig. 7. Time [s] spent on checking restart-robustness w.r.t. each specification and semantics (Color figure online)

Figure 7 compares the runtime of the state-of-the-art tools nuXmv and Z3 on these tasks, with the underlying verification procedures IC3 [11] and Spacer, for the SMV and CHC formalism respectively. The colouring of the marks encodes which restart semantics were considered, while their shape indicates whether the analysed program was elementary or composite.

In our experiments both backends managed to perform all verification tasks in the given time. At a first glance, what strikes the eye, is that nuXmv was about an order of magnitude faster than Z3 on many of the composite examples, while the elementary modules were mostly analysed in less than a second by both tools. However, since this does not apply to all verification tasks for composite modules, compositionality does seem not to be the relevant point here. On closer inspection, we found that in all of the cases where Z3 performed worse, no satisfying interpretation of the CHCs existed. While this is hardly noticeable for the easier tasks, it becomes more apparent in the more complex cases. This might be attributed to nuXmv being more tuned for reasoning over bitvectors, thus quicker to identify unsatisfiable instances, or Spacer not being able to play it's strengths given the non-compositional encoding of program semantics [9,22].

According to the distribution of colours in Fig. 7, the additional consideration of different variants of restart semantics does not seem to have a significant impact on the verification times. The approximate clustering into tasks on elementary and composite programs suggests, that the complexity of the examined program is still the deciding feature. Taken as a whole, the results show that this approach to modelling and verification of restart-robustness w.r.t. some invariant is indeed reasonably fast and feasible.

Results of Synthesis. In this section, we discuss the measurements for our experiments on synthesis of retain configurations that realise restart-robust behaviour w.r.t. a given specification. Since synthesis without retain variables

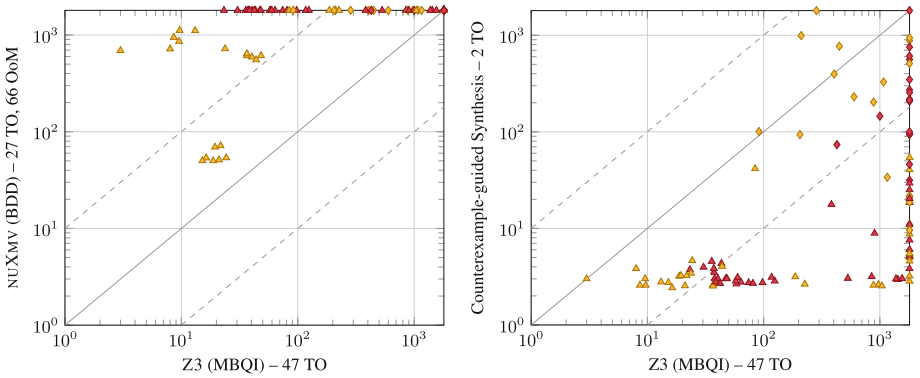

Fig. 8. Time [s] spent on synthesis of restart-robust configuration for each spec and semantics (Color figure online)

does not make sense, we consider the specifications only in the context of delayed or immediate write instrumented and parametrised programs, as seen in Sect. 4.1.

The plots in Fig. 8 illustrate our measurements of the time the verifiers spent on each of the 112 verification tasks, reusing the notation from Fig. 7. To begin with, we focus on the left one, that again compares the runtime of nuXmv and Z3, which now resort to more general decision procedures, i.e. BDD-based CTL checking [27] and a variation of *model-based quantifier instantiation* (MBQI) [19,33] respectively.

Unfortunately, purely BDD-based verification does not scale well for these programs [8], causing nuXmv to run out of memory for 66 verification tasks. In the plot these cases are visualised as timeouts too, i.e. the runtime is set to 1800 s even though the running out of memory occurred earlier. While all synthesis tasks for the composite programs ran out of memory, 27 tasks for the elementary programs caused proper timeouts. Only 19 tasks, all of which targeted elementary modules, could be performed within the resource limits.

Z3 turned out to be significantly more useful for parameter synthesis, timing out only 47 times, and never running out of memory. In contrast to nuXmv, it even manages to determine whether safe retain configurations exist for 14 specifications for the composite programs, and only times out in 11 cases for the elementary ones. We can also observe that, in contrast to plain checking of restart-robustness w.r.t. some property, the type of instrumentation has an impact on the time needed for parameter synthesis. For example, looking at the fastest runs of Z3 we find only yellow triangles, that is tasks on delayed write instrumented elementary modules, ranging from 3 to about 30 s. The corresponding cluster of red triangles, for the immediate write instrumented variants, ranges from about 20 to 150 s.

Nevertheless these results suggest, that the established verification pipelines for checking reachability properties are not practical for parameter synthesis. This observation originally motivated us to devise our own procedure.

Let us now consider the right plot of Fig. 8, which compares the counterexample-guided technique from Sect. 5 with Z3's approach. It is easy to see that our guided approach performs significantly better – often even by more than an order of magnitude. Although it still times out for two of the specifications that refer to the biggest program in our benchmark, the remaining 110 synthesis tasks finished in time. Furthermore, the fact that immediate write instrumented programs yield more complex synthesis problems, does not seem to have a noticeable impact on the runtime. In particular, the clustering of delayed and immediate write instrumented elementary modules that is visible in the x-coordinates, is not apparent in the y-coordinates.

Looking at the plot one might also notice the clustering of data points right above the 2 s mark for our approach. This is due to the fact that we measure the runtime of a verifier from start to finish, i.e. not just the verification procedure, and since our procedure builds upon ARCADE.PLC, the first few seconds of every run are spent on the JVM starting, the PLC program being compiled, and the execution cycle being characterised as a single step.

It is interesting to see that although Z3's approach was worse overall, it managed to analyse one of the cases where our technique timed out. In the end, only one verification task remains unsolved by all approaches.

7 Conclusion

While retain variables were introduced with better safety in mind, they allow for subtle corner cases and unexpected behaviour that only occurs after program restart. We are the first to formalise a logic controller's restart behaviour in the context of delayed or immediate write semantics for retain variables, and approach verification of a program's restart-robustness w.r.t. a specification.

To aid in the design of restart-robust software, we illustrated how synthesis of safe retain configurations can be reduced to verification conditions for existing tooling. We also proposed a counterexample-guided procedure which incrementally approximates a region of safe retain configurations, by exploiting the fact that the actual parameters of the formalisation are Boolean, independent of the retain variables' types. Our experimental results show that the verification conditions for restart-robustness can be solved by established tooling in reasonable time. However, synthesis was only feasible when approached with the counterexample-guided technique.

Future Work. There are several ways in which we want to deepen this line of research. On the one hand we want to examine the feasibility of obvious optimisations, like employing an encoding that allows for compositional reasoning [9,22] or looking into more sophisticated generalisation schemes. This alone might suffice to eliminate the last two timeouts in our benchmark.

On the other hand, we plan on investigating whether a definition of restart-robustness as a relational property between the nominal and restart-augmented behaviour is practical, i.e. given no specification but the program's nominal behaviour, to what extent may the restart-augmented behaviour deviate from it to still be considered robust?

References

1. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2005, Lisbon, Portugal, 5–6 September 2005, pp. 82–87 (2005)
2. Batt, G., Page, M., Cantone, I., Goessler, G., Monteiro, P.T., de Jong, H.: Efficient parameter search for qualitative models of regulatory networks using symbolic model checking. Bioinformatics **26**(18), i603–i610 (2010)
3. Beckert, B., Ulbrich, M., Vogel-Heuser, B., Weigl, A.: Regression verification for programmable logic controller software. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 234–251. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_15
4. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, Austin, Texas, USA, 15–18 November 2009, pp. 25–32 (2009)
5. Biallas, S., Brauer, J., Kowalewski, S.: Arcade.PLC: a verification platform for programmable logic controllers. In: IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, Essen, Germany, 3–7 September 2012, pp. 338–341 (2012)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
7. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
8. Bohlender, D., Hamm, D., Kowalewski, S.: Cycle-bounded model checking of PLC software via dynamic large-block encoding. In: SAC 2018: Symposium on Applied Computing, Pau, France, 9–13 April 2018 (2018, to appear)
9. Bohlender, D., Kowalewski, S.: Compositional verification of PLC software using horn clauses and mode abstraction. In: 14th International Workshop on Discrete Event Systems, WODES 2018, Sorrento Coast, Italy, 30 May–June 1 2018 (2018, to appear)
10. Bohlender, D., Simon, H., Kowalewski, S.: Symbolic verification of PLC safety-applications based on PLCopen automata. In: MBMV 2016, pp. 33–45 (2016)
11. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
12. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22

13. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Parameter synthesis with IC3. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 165–168 (2013)
14. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
16. Darvas, D., Majzik, I., Blanco Viñuela, E.: Formal verification of safety PLC based control software. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 508–522. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_32
17. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, Austin, TX, USA, 30 October–02 November 2011, pp. 125–134 (2011)
18. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
19. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
20. Hauck-Stattelmann, S., Biallas, S., Schlich, B., Kowalewski, S., Jetley, R.: Analyzing the restart behavior of industrial control applications. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 585–588. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_38
21. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
22. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD 2015, pp. 89–96 (2015)
23. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 846–862. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_59
24. Koskinen, E., Yang, J.: Reducing crash recoverability to reachability. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 97–108 (2016)
25. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-540-74105-3
26. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. In: Broy, M. (ed.) Program Design Calculi, vol. 118, pp. 287–323. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-662-02880-3_10
27. McMillan, K.L.: Symbolic Model Checking. Kluwer, Dordrecht (1993)

28. Moon, I.: Modeling programmable logic controllers for logic verification. IEEE Control Syst. **14**(2), 53–59 (1994)
29. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
30. Ovatman, T., Aral, A., Polat, D., Ünver, A.O.: An overview of model checking practices on verification of PLC software. Softw. Syst. Model. **15**(4), 937–960 (2016)
31. PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, Germany (2006)
32. PLCopen TC5: Safety Software Technical Specification, Version 1.01, Part 2: User Examples. PLCopen, Germany (2008)
33. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods Syst. Des. **42**(1), 3–23 (2013)