Test Suite Augmentation for Reconfigurable PLC Software in the Internet of Production

Marco Grochowski(✉), Marcus Völker, and Stefan Kowalewski

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

Abstract. Regression testing is an established technique used to attest the correctness of reconfigurations to PLC software. After such a reconfiguration, a test suite might not be adequate to ensure the absence of regressions, requiring the derivation of new test cases to uncover potential regressions. This paper presents a combination of state-of-the-art symbolic execution algorithms for test suite augmentation, an indispensable part of regression testing. Test generation is guided towards the changed behavior using a technique known as four-way forking. The old and new PLC software are executed in the same symbolic execution instance to account for the effects of the reconfiguration and increase the chances of generating difference-revealing test cases. The prototypical implementation is evaluated using domain-specific benchmarks such as the PLCopen Safety library and the Pick and Place Unit, exposing the limitations in applicability and effectiveness of the used techniques for safeguarding PLC software subject to frequent reconfigurations as found in cyber-physical production systems.

Keywords: Regression testing · Test suite augmentation · Symbolic execution · Programmable logic controllers · Internet of Production

1 Introduction

Transformability, a property resulting from the flexibility and mechanical reconfigurability of a cyber-physical production system (CPPS), is one of the primary enablers to cope with changing intrinsic and extrinsic demands and is a necessary prerequisite to guarantee the ability to compete with other companies [9]. An overview of the life cycle and value chain of a CPPS is given in Fig. 1. In contrast to a conventional production system, a CPPS is subject to a high degree of reconfigurability during its life cycle. This highly agile manufacturing paradigm leads to an increase in complexity as the insights gained during production turns into data that controls the production process. Due to the heterogeneity and emergent behavior of CPPS, unwanted regressions might accompany those reconfigurations and take their toll on the functional safety and reliability of software components [6]. In the context of static reconfigurations where the entire CPPS

J. F. Groote and M. Huisman (Eds.): FMICS 2022, LNCS 13487, pp. 137–154, 2022.
https://doi.org/10.1007/978-3-031-15008-1_10

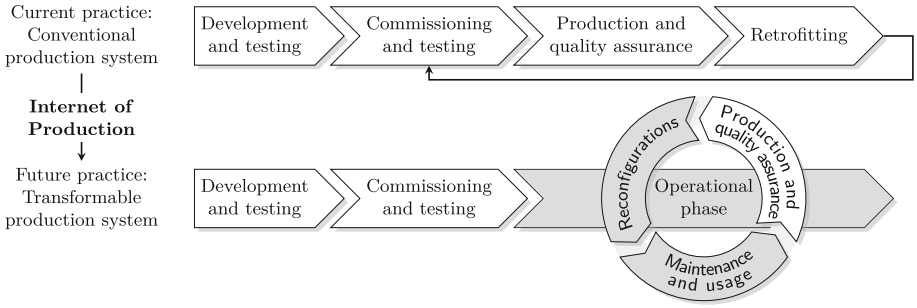

Fig. 1. Juxtaposition of the life cycle and value chain of cyber-physical production systems (Figure adapted from illustration in [19]).

is stopped and analyzed during maintenance, short downtimes are crucial, and we argue that lightweight verification techniques such as testing are suitable to assess the CPPS's correctness quickly. Consequently, the goal is to reduce the lead-time after a reconfiguration to the CPPS has occurred by reducing the time it takes to test the reconfigured programmable logic controller's (PLC) software throughout the ramp-up phase during maintenance as depicted in Fig. 1.

Regression Testing. Regarding the reconfigurations to PLC software, they manifest themselves in the form of the addition of new functionality, the modification of already existing functionality, or the removal of functionality, which most often also requires adaptations to the test suite. As the manual creation of difference-revealing test cases requires enormous effort and expertise, automated techniques are desirable. One prominent set of such automated techniques that tackles test suite maintenance is termed regression testing. Figure 2 illustrates the process of regression testing and test suite augmentation after a syntactic reconfiguration. Consider the test suite T^P_{all} for a PLC program P before a reconfiguration with which the reconfigured PLC program P' should be tested. There are two primary reasons why re-executing the whole test suite is infeasible. The first one is that the test suite might be too large and require too much time while not focusing on the parts of the software affected by the reconfiguration. The other aspect is that the test suite might not even test the changed behavior of P'. In this sense, test suite augmentation is necessary and an important complementary technique to traditional regression testing techniques [21,23]. Dealing with reconfigurations to the PLC software and its effect on the test suite is a two-step approach during test suite maintenance. First, one has to assess if the test suite T^P_{all} is still *adequate* enough for testing P'. Standard measures for adequacy are whether the test suite is homogeneous with regards to the program paths, for instance, line or branch coverage. Nonetheless, one has to keep in mind that coverage alone does not quantify the capability of a test suite to reveal regressions. If the test suite is not homogeneous with regards to the failure [20], i.e., it structurally covers the reconfigured program path but does not propagate a divergence to the output, it will not reveal the regression after a faulty syntactic

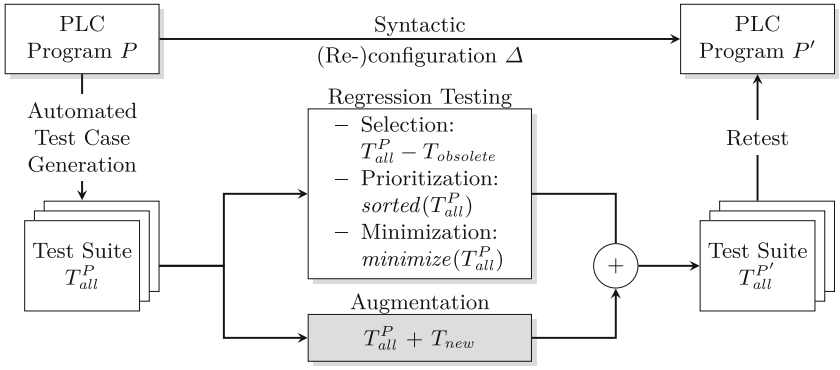

Fig. 2. Application of regression testing techniques and test suite augmentation after a syntactic reconfiguration.

reconfiguration. Second, the reconfigurations in P' need to be identified, and the test case generation algorithm has to be guided to cover the potentially reconfigured behavior. As regressions are only observable for inputs that expose a behavioral difference, we use a concept coined as four-way forking [10] to guide the test case generation into parts of the software affected by a reconfiguration. As the identification of the reconfiguration is a challenging problem, we resort to manual software annotations to explicitly denote the reconfigured parts from one version to another.

Syntactic Reconfiguration. The syntactic reconfiguration mentioned in Fig. 2 follows the concept presented in [10], where a `change(old,new)` macro was used to characterize the effect of the reconfiguration. The first argument of this macro represents the expression from the PLC software before the reconfiguration, and the second argument represents the expression of the PLC software after the reconfiguration. As a result, the manifestation of reconfigurations to PLC software stated earlier, i.e., the addition of new functionality, e.g., adding an extra assignment $x \coloneqq \texttt{change}(x, 1);$, the modification of already existing functionality, e.g., changing the right-hand side of an assignment $x \coloneqq y + \texttt{change}(1, 2);$, or the deletion of functionality, e.g. removal of straightline code $\text{if}(\texttt{change}(\text{true}, \text{false})) \ldots \text{code} \ldots$ can be expressed succinctly with the `change(old,new)` macro. This way of annotating the expressions of reconfigured parts of the software has a significant benefit as it keeps the correspondence between both versions intact and was therefore chosen for analyzing the semantic effects of the implication introduced by the reconfigurations.

1.1 Limitations and Contributions

A premise resulting from the introduction is the existence of syntactically change-annotated PLC programs given as input to our framework. To further narrow the

scope of this contribution, the peculiarities of PLCs have to be considered. A PLC is subject to cyclic execution resulting in non-termination. Still, every execution through one cycle terminates and hence can be analyzed. The programming languages for PLCs forbid recursive calls, i.e., the call-flow graph is acyclic [8]. Furthermore, our framework does not support the use of arrays or pointers yet. Nevertheless, statically allocated memory can be modeled by flattening the arrays. While the prototypical framework is able to analyze loops other than the naturally occurring execution cycle of the PLC program, these loops are not explicitly handled and analysis might be intractable. As some of the benchmarks use the timer capabilities of the IEC 61131-3 standard [8], we use an over-approximating representation of timers from [1], which non-deterministically models the internal decision variable measuring the passing of time. Last but not least, control tasks are usually distributed in the context of Industry 4.0, yet most often still coordinated centrally. Instead of having a single PLC that controls the various actuators in the CPPS, multiple PLCs exist, one for each control task and one overarching, coordinating PLC. Despite that, we model the distributed control task as one, compositional, *classic* PLC program, in which the other control tasks are incorporated as function blocks and executed on one single PLC controller (cf. Sect. 4). This neglects the influences of different times and latencies introduced due to the communication between each controlling PLC. We assume that the sequential modeling using a single PLC is a feasible abstraction of several distributed PLCs running in parallel, realizing the same control task, because the business logic is implemented by a single, coordinating PLC, which processes the messages of the other distributed PLCs sequentially in all circumstances. To this end,

- we improve the scalability of an existing Dynamic Symbolic Execution (DSE) algorithm for PLC software,
- we evaluate the feasibility of DSE and the concept of four-way forking for test suite augmentation of reconfigured PLC software on benchmarks of varying difficulty and compare it to previous results.

2 Related Work

Symbolic execution is one of the primary techniques for software testing and resulted in the development of numerous language-agnostic analysis tools in the past [3]. Previous work has investigated the applicability of DSE in test suite generation for PLC software [4]. The results were promising but have not been applied to tackle the problem of test suite augmentation after a reconfiguration. In contrast to [4], the concolic and compositional DART algorithm, also known as SMART [5], explores the program execution tree depth-first on a per path basis allowing for the use of summaries. However, we currently refrain from summarization due to our conflicting merging strategy. An approach that aids regression testing with static change-impact analysis is called directed incremental SE (DiSE) [22]. The rationale behind this is that static analysis avoids the problems of undecidability of whether there exists an input that is difference-revealing against the reconfigured program by over-approximating the semantic properties using syntactic dependencies such as control- and data dependencies.

The results from the static analysis are used to guide symbolic execution by exploring only paths that can reach parts of the software affected by the reconfiguration. This approach, however, has two severe disadvantages. We argue that these *slices* give only conservative estimates and are often too imprecise, reducing opportunities for information reuse from the prior analysis of the reconfigured PLC software. Furthermore, DiSE only explores one execution path through the impacted parts of the software, and besides reachability, there is no guidance in the direction of real divergences. This lead us to the choice of Shadow Symbolic Execution (SSE) [10] for test suite augmentation. SSE uses a seeded exploration with a test case that touches the presumable *patch*, or in our terminology, the reconfiguration. The novelty of SSE is that it executes the old (presumed buggy) and new (presumed patched) program within the same SE instance. Therefore, it allows the algorithm not to re-execute potentially expensive path prefixes, which provides several opportunities to prune and prioritize paths and simplify constraints. Despite this, the reconfigurations are touched by a test case that dictates the context in which the potential reconfiguration is reached and hence limits the generalization. Furthermore, both programs need to be merged into a change-annotated, unified version.

Verification and Testing in the PLC Domain. Regarding the safeguarding of reconfigurations in the PLC domain several techniques on various levels have emerged in the past years. TestIas [24] is a tool for model-based verification of reconfigurations to distributed automation systems. It works on a higher level than PLC software, i.e., trying to prove the correctness of a reconfiguration affecting the functional perspective of services in a CPPS. Prioritization for regression testing of reconfigured PLC software with regards to system tests was evaluated in [17]. It optimizes the regression testing process of CPPS after a reconfiguration, however, it is unable to generate difference-revealing test cases. Another interesting approach poses the modular regression verification of the VerifAps library which was successfully applied to the Pick and Place Unit (PPU) case study in [18]. Modular regression verification requires the specification of relational regression verification contracts allowing for the decomposition of the verification task resulting in efficient solving, yet being far from a push-button technology.

3 Methodology

An overview of our prototypical test suite augmentation (TSA) framework is given in Fig. 3 and explained throughout this section. TSA can be considered as a development time technique, in which the developer manually annotates the desired changes and is able to assess their implications on the observable behavior of the PLC software. The input to the program analysis framework is a manually change-annotated PLC program in structured text (ST), one of the five IEC 61131 programming languages [8], using the `change(old,new)` annotation macro introduced in Sect. 1. Before going in-depth with the core TSA algorithm, we briefly describe our intermediate representation of the PLC software.

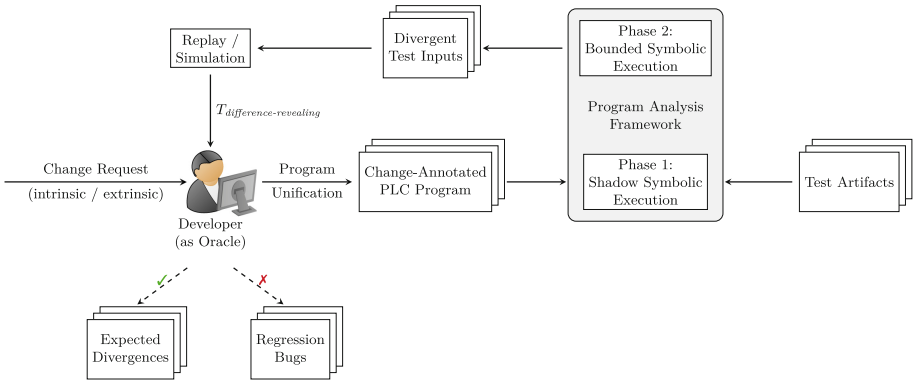

Fig. 3. Overview of the prototypical TSA framework.

3.1 Intermediate Representation

A PLC program can consist of several program organization units (POUs), which provide an interface definition of the input, local, and output variables, and a body containing the actual instructions that operate on this interface. The IEC 61131 standard [8] distinguishes between three types of POUs, namely functions, function blocks, and programs. A program represents the main entry, whereas function blocks and functions represent stateful and stateless procedures, respectively. At cycle entry, new input values are read from the environment and written to the input variables. During the execution of the cycle, the program operates on a copy of these input variables and internal state variables. The state variables also comprise output variables written to the PLC's output at the cycle exit. While new values are assigned to input variables in each cycle, the internal state variables retain their values. During the parsing and compiling of the input program, function blocks are lowered to *regular* procedures operating on references of their variables. As a result, parameterized calls to function blocks are modeled as parameterless calls preceded and succeeded by a sequence of input and output assignments in the respective caller, which do not modify the state explicitly but rather transfer the flow of control between procedures. For this purpose, we have chosen a goto-based intermediate representation (IR) to represent a subset of the ST language [8] in form of a so-called control-flow graph (CFG) [2]. We model the PLC program as a pair $P = (G, \mathcal{G})$, where $G \in \mathcal{G}$ is the CFG of the program POU, and $\mathcal{G}$ is a set of CFGs representing nested function blocks occurring in the program. The instructions supported by this IR are defined over variables $x \in \mathbf{X}$, Boolean and arithmetic expressions e as usual

$$I ::= \mathbf{assign}(x, e) \mid \mathbf{ite}(e, \mathbf{goto}\ b_{\ell_1}, \mathbf{goto}\ b_{\ell_2}) \mid \mathbf{call}\ G' \mid \mathbf{return} \mid \mathbf{cycle}\ .$$

Unlike in typical goto-based IRs, we introduced a **cycle** instruction, explicitly denoting the end of the execution cycle. Given the terminology, we will dive into the baseline symbolic execution framework used for generating the test cases which is reused during the application of SSE.

3.2 Bounded Symbolic Execution

Our implementation of the Bounded Symbolic Execution (BSE) for TSA is composed of three components: an execution context, an executor, and an exploration strategy. An execution context $q = (c, \ell, f, \rho, \sigma, \pi)$ consists of a cycle c, a label ℓ referring to a vertex b_ℓ of a CFG G, a frame stack f, a concrete store ρ, which associates variables with concrete values, a symbolic store σ, which associates variables with symbolic values, and a path constraint π. The frame stack f holds triples $(G_{callee}, scope, \ell_{caller})$, where G_{callee} denotes the CFG of the callee, *scope* is the scope in which the call occurred, and ℓ_{caller} denotes the intra-procedural label of the caller at which the execution should resume after returning from the callee. The BSE algorithm is given in Algorithm 1 and explained in the following. It is also commonly known as compositional SE in literature [3] augmented with parameterizable local and global termination criterias.

Exploration Strategy. We decided for a cycle-based, depth-first exploration strategy similar to [4] with parameterizable timeout, coverage, and cycle bounds. As the cyclic execution of PLC programs significantly increases the computation time of symbolic execution, we adjusted the termination criteria in line 2 to consider a configurable cycle exploration bound. The priority queue $\mathcal{Q}$ is sorted heuristically by prioritizing execution contexts with a lower cycle count, resulting in the exploration of all feasible execution paths through one execution cycle before continuing with the next cycle. Furthermore, candidate execution contexts with a deeper path length and a concretely executable store are prioritized over execution contexts with a shallower path length. This enables the depth-first exploration to simulate a breadth-first exploration through one cycle and generates concise test cases with no unnecessary executed cycles. When encountering the end of the cycle during execution (cf. line 25), the cycle counter is increased and new concrete input valuations and fresh symbolic variables are derived.

Assignments, Branches and Calls. The semantic effects of the instructions on the respective stores are captured via an evaluation function `eval`. For an assignment $\mathbf{assign}(x, e)$, the concrete and symbolic store are updated via $\rho \leftarrow \rho[x \mapsto \texttt{eval}_\rho(e)]$ and $\sigma[x \mapsto \texttt{eval}_\sigma(e)]$, respectively, as illustrated in line 10. The bracket notation [] denotes the usual replacement for the specified variable in the store. Whenever an $\mathbf{ite}(e, \mathbf{goto}\,\ell_1, \mathbf{goto}\,\ell_2)$ instruction is encountered, the path constraint is updated symbolically depending on the result of the branch expressions concrete evaluation (cf. line 12). In case the expression evaluates to true, execution is continued in the positive branch and a test case is derived if this label is yet uncovered. We also check if the other path is feasible under the current path constraint and fork the execution context with the concrete valuation of the model (cf. lines 15–19). As mentioned in the beginning of Sect. 3.1, call and return effects are lowered to input and output assignments during compilation. Therefore, the **call** and **return** instruction modify the frame stack and update the control-flow accordingly.

Algorithm 1: Bounded Symbolic Execution

Input : Program $P = (G, \mathcal{G})$, CFG $G = (\mathbf{X}, \mathbf{X}_{in}, (B, E), b_{\ell_e}, b_{\ell_x})$
Output : Test Suite T

1 $\mathcal{Q} \leftarrow \{(0, \ell_e, \emptyset, \rho_{\ell_e}, \sigma_{\ell_e}, \mathit{true})\}$; $\mathcal{M} \leftarrow \emptyset$
2 **while** $\big(\mathcal{Q} \neq \emptyset \vee \mathcal{M} \neq \emptyset\big) \wedge \neg$`terminationCriteriaMet` **do**
3 **if** $\mathcal{Q} = \emptyset$ **then** $\mathcal{Q}$.`push`(`merge`($\mathcal{M}$))
4 $q \leftarrow (c, \ell, f, \rho, \sigma, \pi) \leftarrow \mathcal{Q}$.`pop`()
5 **if** `reachedMergePoint`(q) **then**
6 $\mathcal{M}$.`push`(q)
7 **else**
8 **switch** `instructionAt`(ℓ) **do**
9 **case assign**(x, e) **do**
10 $\mathcal{Q}$.`push`$\big((c, \ell + 1, f, \rho[x \mapsto \texttt{eval}_\rho(e)], \sigma[x \mapsto \texttt{eval}_\sigma(e)], \pi)\big)$
11 **case ite**(e, **goto** ℓ_1, **goto** ℓ_2) **do**
12 **if** $\texttt{eval}_\rho(e)$ **then**
13 $q_1 \leftarrow (c, \ell_1, f, \rho_1, \sigma_1, \pi \wedge \texttt{eval}_\sigma(e))$; $\mathcal{Q}$.`push`(q_1)
14 **if** $\neg$`covered`(ℓ_1) **then** T.`deriveTestCase`(q_1)
15 **if** `tryFork`($\pi \wedge \texttt{eval}_\sigma(\neg e)$) **then**
16 $\rho_2 \leftarrow$ `model`($\pi \wedge \texttt{eval}_\sigma(\neg e)$))
17 $q_2 \leftarrow (c, \ell_2, f, \rho_2, \sigma, \pi \wedge \texttt{eval}_\sigma(\neg e))$; $\mathcal{Q}$.`push`(q_2)
18 **if** $\neg$`covered`(ℓ_2) **then** T.`deriveTestCase`(q_2)
19 **end**
20 **else** `// analogous, omitted for brevity`
21 **case call** G' **do**
22 f.`push`(G', `getScope`(G), ℓ_{G_r}); $\mathcal{Q}$.`push`$\big((c, \ell_{G'_e}, f, \rho, \sigma, \pi)\big)$
23 **case return do**
24 $(\_, \_, \ell_{G_r}) \leftarrow f$.`pop`(); $\mathcal{Q}$.`push`$\big((c, \ell_{G_r}, f, \rho, \sigma, \pi)\big)$
25 **case cycle do**
26 $\mathcal{Q}$.`push`$\big((c + 1, \ell_e, f, \rho[x \in \mathbf{X}_{in} \mid x \mapsto \texttt{random}()], \sigma[x \in \mathbf{X}_{in} \mid x \mapsto x_{fresh}], \pi)\big)$
27 **end**
28 **end**
29 **end**
30 **return** T

Merge Strategy. Execution contexts are merged at all possible points where the control-flow joins with respect to realizable paths as opposed to merging at the cycle end as in [4]. During execution, we check whether the current context reached an interprocedural realizable merge point (cf. line 5) and add it to the merge queue $\mathcal{M}$ for further processing.

Unreachable Branches. The detection of unreachable branches is an essential task to avoid the encoding of infeasible paths when applying symbolic execution. As our static analysis (SA) is currently not capable of abstract interpretation, we leveraged the algorithms from Crab[1] to build a value set analysis calculating the

[1] https://github.com/seahorn/crab.

possible values for each variable at each label of our CFGs. Using this information, we can statically deduce whether a branch is reachable or not. While being a powerful tool it is apparent that the SA of CRAB is not tailored to the domain of PLC software. To express our IR in a form such that CRAB is able to analyze it, it passes several code transformation pipelines including basic block encoding, three-address code, call-transformation and static single assignment which severely bloats up the CFG representation. In order to get precise information the expensive boxes domain was chosen [7]. The boxes domain is sensitive to the number of "splits" on each variable which come, among other things, from joins and Boolean operations. Unfortunately, the benchmarks in Sect. 4 "split" a lot due to the cyclic dependency between variables and the state-machine like behavior. Therefore, to still be able to reuse at least some information from the SA, we decided for a trade-off between precision and run time by tuning the behavior of the boxes domain to convexify after a certain amount of disjunctions resulting in imprecise but still usable results.

3.3 Shadow Symbolic Execution

Intuitively, two things are needed for TSA after a reconfiguration: (1) the test cases must reach potentially affected areas along different, relevant paths (specific chains of data- and control-dependencies), and (2) test cases must account for the state of the PLC software and the effects of the reconfigurations, i.e., be difference-revealing. An interesting research question in this context is whether the concept of four-way forking stemming from the SSE [10] algorithm is applicable to the PLC domain using the `change(old, new)` macros (cf. Sect. 1) to apply TSA for reconfigurable PLC software. In general, it can be intractable, because outputs are potentially difference-revealing after k cycles (depending on the internal state) and hence the analysis runs out of memory before the difference is reached. In general, deriving difference-revealing test cases in the style of SSE [10] is a two-step application of SE algorithms (cf. Fig. 3) and is presented in detail in Algorithm 2. In line 1 of Algorithm 2 the test suite of the version before the reconfiguration is reused and executed on the change-annotated PLC program to determine which test cases "touch" the change. Prior to execution, in case the interface has changed due to the

Algorithm 2: Test Suite Augmentation using SSE [10]

Input : Program $P = (G, \mathcal{G})$, CFG $G = (\mathbf{X}, \mathbf{X}_{in}, (B, E), b_{\ell_e}, b_{\ell_x})$, Test Suite $\mathcal{T}$

Output : Difference revealing test cases $T_{\text{difference-revealing}}$

1 $T_{\text{change-traversing}} \leftarrow$ `collectChangeTraversingTestCases(`$G, \mathcal{T}$`)`
2 **foreach** $t \in T_{\text{change-traversing}}$ **do** `// Phase 1 - SSE`
3 $\quad \{(q_0, t'_0), \ldots, (q_m, t'_m)\} := \mathcal{Q}_{\text{divergent}}$`.push(findDivergentContexts(`$t$`))`
4 **end**
5 **foreach** $(q, t') \in \mathcal{Q}_{\text{divergent}}$ **do** `// Phase 2 - BSE`
6 $\quad T_{\text{divergent}}$`.push(performBoundedExecution(`$q, t'$`))`
7 **end**
8 $T_{\text{difference-revealing}} \leftarrow$ `checkForOutputDifferences(`$T_{\text{divergent}}$`)`

reconfiguration, the test case does not contain valuations for all variables. Therefore, we augment the test case with additional valuations using the 0-default initialization for *BOOL* and *INT* as defined in IEC61131-3, *false* and 0, respectively. Each executed test case is further augmented with additional information such as the execution history and state valuations reaching the end of the cycles of the old program version. As a test case can "touch" multiple change-annotated labels, we consider only the test cases that cover as much information as possible with regards to the respective change-annotated label. This reduces the amount of test cases needed for consideration in the first phase without losing expressiveness, as test cases spanning along multiple cycles with the same prefix are prioritized. When functionality is added depending on newly introduced input variables, the prior test suite is unable to cover these labels, hence we keep track of labels that were change-annotated but not "touched" by any test case.

Finding Divergent Contexts. Before continuing with the explanation of Algorithm 2, we present how divergent contexts are found during symbolic execution. Algorithm 3 uses the concept of four-way forking to determine whether the execution of a test case leads to potential divergent behavior or not. It is driven by the concrete input valuations of the corresponding test case (cf. line 1) and the augmented BSE is concolically executed on a per cycle basis using a single execution context, hence no merging. In general, the algorithm is similar to the one presented in Algorithm 1. We adapted the handling of branches to support the four-way forking and introduced additional data structures for storing the shadow expressions in the context, here hidden behind the concrete and symbolic store. As change annotations may occur in any instruction (or expression) we use the notion of symbolic change shadows and check whether such a change shadow influences the behavior of the current execution path. In case a branch is encountered during the concolic execution of the test case, we recursively check if the expression contains a symbolic change *shadow* (cf. line 7). If the current branch expression contains no shadow expression, we continue the execution as illustrated in Algorithm 1 in the lines 12–20. In case the branch expression contains a shadow expression, it might lead to divergent behavior. In order to check whether the current test case takes different paths in the old and the new version of the code, we first evaluate it under the concrete store of the divergent context resolving all shadow expressions (cf. line 8). If the valuations of the expression in the old and the new context do not coincide, the test case exposes truly divergent behavior which might trigger difference-revealing outputs. At this point, the execution stops and the divergent context is added to the queue to be explored in the second phase. If the valuations are equal, there still might be potential divergent behavior. First, we encode the expression using the old and the new symbolic valuations and then check in lines 14–17 whether potential divergent behavior exists. For this purpose, we explore subsequently whether there exist concrete valuations that may diverge and derive a test case as a witness. The forked divergent context is added to the divergent context queue and the execution continues with either following the true or the false branch trying to propagate the execution context to a deeper nested potentially divergent

Algorithm 3: findDivergentContexts – BSE with four-way forking [10]

```
   Input  : CFG G = (X, X_in, (B, E), b_{ℓ_e}, b_{ℓ_x}), Test Case t
   Output : Divergent Contexts Q_divergent
 1 foreach c_t ∈ t do
 2     q ← (c, ℓ, f, ρ^{c_t}_input, σ, π)
 3     while c = c_t do
 4         switch instructionAt(ℓ) do
 5                         // other cases analogous to BSE, omitted for brevity
 6             case ite(e, goto ℓ_1, goto ℓ_2) do
 7                 if containsShadowExpression(eval_σ(e)) then
 8                     (v_old, v_new) ← eval^{shadow}_ρ(e)
 9                     if v_old ≠ v_new then                    // divergent behavior
10                         Q_divergent.push((q, t))
11                         return Q_divergent
12                     else                                // potential divergent behavior
13                         (φ_old, φ_new) ← eval^{shadow}_σ(e)
14                         if tryDivergentFork(π ∧ ¬φ_old ∧ φ_new) then
15                             Q_divergent.push((q_forked, deriveTestCase(q_forked)))
16                         end
17                         if tryDivergentFork(π ∧ φ_old ∧ ¬φ_new) then
18                             Q_divergent.push((q_forked, deriveTestCase(q_forked)))
19                         end
20                         if v_old then
21                             q ← (c, ℓ_1, f, ρ, σ, π ∧ φ_old ∧ φ_new);
22                         else
23                             q ← (c, ℓ_2, f, ρ, σ, π ∧ ¬φ_old ∧ ¬φ_new);
24                         end
25                     end
26                 else                        // analogous to BSE, omitted for brevity
27                 end
28             end
29         end
30 end
31 return Q_divergent
```

context. On termination, i.e., either when a divergent context is found or when all the concrete input valuations for each cycle of this test case have been executed, the algorithm returns the set of divergent contexts and continues with the next test case.

Propagating Divergent Contexts. The second phase of Algorithm 2 performs a seeded BSE (cf. Algorithm 1) for each found divergent context in the first phase. The divergent context and test case passed as parameters in line 6 represent either a diverging concrete execution or were generated because of a potential, possible divergence at the four-way fork in the first phase. This phase runs until the

termination criteria is met and tries to generate as many test cases as possible. These test cases cover paths originating from a divergence and hence may expose differences in the outputs between the old and the new version of the reconfigured PLC program. In line 8 of Algorithm 2 the derived divergent test cases are checked for output differences. The execution of modified instructions does not mean that they are necessarily difference-revealing because the subdomains do not need to be homogeneous with regards to the failure [20]. Hence to determine whether a test case exposes an externally observable difference, the outputs on the test case in the new version are compared to the outputs on the test case in the old version. If the outputs differ on a per cycle basis, the test case is added to the set of difference-revealing test cases and requires further examination by the developer.

4 Evaluation

The evaluation was conducted on an Intel(R) Core(TM) i5-6600K CPU @ 3.50 GHz desktop with 16 GB of RAM running openSUSE Leap 15.3. For SMT-solving, we utilized the high-performance automated theorem prover Z3 by Microsoft [13]. The benchmarks evaluated with ARCADE.PLC were also run with the same evaluation setup. The code of our contribution and the corresponding benchmarks are available for download at https://github.com/embedded-software-laboratory/TSA-FMICS22.

In the following, we first present the achieved performance improvements for the BSE as our TSA implementation heavily relies on it before presenting the results of the TSA algorithm on a few selected benchmarks.

PLCopen Safety Suite. The benchmark consists of a set of safety-related PLC programs provided by the PLCopen organization [15]. The results are listed in Table 1 and show for each evaluated function block the lines of code (LOC), the coverage values as well as the runtimes of the implementation of a merge-based test generation algorithm in ARCADE.PLC [4] in comparison to the results of our contribution. Because both tools use different IRs, the number of reachable branches is omitted. The timeout (TO) was set to 10 min. For the detection of unreachable branches, ARCADE.PLC uses a values-set analysis, however, we did not add the time to the results. Instead, we ran both programs with this additional pre-computed information to only focus on the performance of the DSE algorithms. The SA_{manual} refers to the use of CRAB and manual annotation for truly unreachable branches which were over-approximated due to the convexification of the disjunctions (cf. Sect. 3). As far as the function blocks are concerned, both approaches perform equally well. As all blocks follow the same general structure, the LOC can be seen as a reference for giving a rough estimate on what one would expect time wise from the analysis. A significant difference between both approaches is the amount of test cases generated. While ARCADE.PLC generates concise test cases for every branch, our contribution tries to avoid redundancies due to shorter test cases being included in longer test cases, hence generating less test cases overall. This is neither a benefit nor a disadvantage and could be obtained

by a static postprocessing on the test suite generated by ARCADE.PLC. Do note that ARCADE.PLC does not dump any test cases in case it runs into a TO due to a technical limitation. The programs on the bottom half are bigger in the sense that they are composed of multiple function blocks from the top with additional logic and were analyzed without manual SA annotation. As more and more calling contexts are available it becomes apparent that delaying the merging until the end of the cycle performs way worse than merging on all realizable paths when the opportunity emerges. Most notably, the performance degenerates on blocks which make heavy use of timer and edge trigger function blocks because only specific paths can reach deeper behavior.

Table 1. Comparison of branch coverage and runtimes for the test generation of the PLCopen Safety library, ordered alphabetically.

Function Block / Program		ARCADE.PLC + SA			Contribution + SA_{manual}		
	LOC	**cov. [%]**	**T [#]**	**time [s]**	**cov. [%]**	**T [#]**	**time [s]**
`Antivalent`	136	100	61	0.74	100	23	0.37
`EDM`	229	100	134	5.22	100	62	3.49
`Emergency_Stop`	127	100	66	0.45	100	27	0.33
`Enable_Switch`	133	100	71	1.13	100	32	1.28
`Equivalent`	133	100	62	0.86	100	26	0.59
`ESPE`	127	100	66	0.42	100	27	0.31
`Guard_Locking`	148	100	80	1.01	100	37	0.87
`Guard_Monitoring`	174	100	82	1.45	100	34	1.12
`Mode_Selector`	239	100	70	5.20	100	30	1.08
`Muting_Seq`	262	97.5	-	TO	100	53	49.6
`Out_Control`	121	100	67	0.77	100	31	0.61
`Safe_Stop`	157	100	73	3.52	100	32	0.59
`Safely_Limit_Speed`	175	100	91	9.90	100	41	1.38
`Safety_Request`	191	100	88	1.29	100	40	1.01
`Testable_Safety_Sensor`	291	100	147	16.93	100	68	17.08
`Two_Hand_Control_Type_II`	126	100	83	0.85	100	38	0.73
`Two_Hand_Control_Type_III`	184	100	107	1.63	100	46	0.95
`DiagnosticsConcept`	537	65.49	-	TO	91.00	104	TO
`Muting`	1119	51.24	-	TO	80.23	196	TO
`SafeMotion`	1061	38.15	-	TO	73.71	156	TO
`SafeMotionIO`	811	53.50	-	TO	71.65	106	TO
`TwoHandControl`	608	58.79	-	TO	86.34	131	TO

Pick and Place Unit (PPU). The benchmark consists of a total of 15 scenarios for the PPU of an open-source bench-scale manufacturing system[2]. While it is

[2] https://www.mw.tum.de/ais/forschung/demonstratoren/ppu/.

limited in size and complexity, this trade-off between problem complexity and evaluation effort does not harm the expressiveness of the benchmark. In this evaluation, we focused on the first four scenarios and translated them from their PLCopen XML representation to ST using the VerifAps library[3]. The `Scenario_0` consists of a stack, crane and a ramp of which the latter is only mechanical. The reconfiguration `Scenario_`$\{0 \rightarrow 1\}$ aims to increase the ramp's capacity. This reconfiguration does not affect the software as the ramp is a purely mechanical component. As a response to changing customer requirements, the reconfiguration `Scenario_`$\{0 \rightarrow 2\}$ enables the PPU to handle both plastic and metallic workpieces. For this purpose, an induction sensor is introduced which changes the output behavior of the stack component. The behavior of the crane is untouched. The third reconfiguration `Scenario_`$\{2 \rightarrow 3\}$ introduces the stamping functionality of metallic workpieces. This impacts the behavior of the crane as workpieces need to be stamped before being transported to the ramp. The results of the test suite generation using BSE without SA results are shown in Table 2. The PPU has more complex behavior in comparison to the PLCopen safety suite, which is also reflected in the required time/termination criteria for the test case generation. A comparison with Arcade.PLC was omitted as it was not able to analyze the benchmarks.

Table 2. Results of the test suite generation using BSE for selected PPU scenarios.

PPU Scenario	**LOC**	Contribution			
		cov. [%]	**T [#]**	**time [s]**	**cycle [#]**
`Scenario_0`	412	88.97	45	169.82	25
`Scenario_1`	412	88.97	45	170.12	25
`Scenario_2`	459	89.61	55	274.19	25
`Scenario_3`	768	91.67	102	1198.08	25

Table 3 shows the results of the TSA for the manually change-annotated reconfigured PLC programs.

Table 3. Results of the TSA using Algorithm 2 for selected reconfiguration scenarios of the PPU.

PPU Evolution	ℓ_{ca} [#]/ ℓ_u [#]	T_{ca} [#]	Phase 1		Phase 2		T_{diff} [#]
			$\mathcal{Q}_{div}$ [#]	**t [s]**	T_{div} [#]	**t [s]**	
`Scenario_`$\{0 \rightarrow 1\}$	0/0	0	0	0	0	0	0
`Scenario_`$\{0 \rightarrow 2\}$	12/1	45	2	1.77	52	54.99	23
`Scenario_`$\{2 \rightarrow 3\}$	50/21	55	21	19.49	1269	3423.94	1269

[3] https://github.com/VerifAPS/verifaps-lib.

The first column of Table 3 denotes the analyzed reconfiguration scenario. The second column contrasts how many change-annotated labels ℓ_{ca} in the reconfigured program exists and how many of those change-annotated labels remain untouched ℓ_u by the test suite of the prior version. This ratio gives an estimate on how suited the previous test suite is to find divergences. The third column denotes the number of test cases T_{ca} in the previous test suite which exercise any number of change-annotated labels ℓ_{ca} in the change-annotated PLC program. One has to keep in mind that the generated test cases are succinct with regards to the required number of cycles to reach a specific branch (in case of branch coverage). Due to the cyclic nature of the PLC software, test cases which cover deeper nested branches, i.e., branches reachable after a certain amount of cycles, can share a partial prefix with test cases covering already some of the branches on these paths. This is a natural limitation of the SSE approach for cyclic programs resulting in an increased analysis time for phase 1 and phase 2. The fourth column denotes the number of derived divergent contexts and the time it took to complete phase 1 for each representative test case. The fifth column denotes the number of divergent test cases generated from propagating the divergent contextes during BSE using the corresponding triggering test cases as a seed for the concolic execution and the time it took to complete phase 2. The sixth column denotes the number of difference-revealing test cases found by checking the observable behavior of the old and the new version of the program on the divergent test cases.

5 Conclusion

The state of the art for TSA is dominated by DSE techniques [3]. We implemented a baseline BSE improving scalability issues prevalent in prior work [4] due to infrequent merging and inefficient storing of the execution contexts. On top of this baseline, we implemented the concept of four-way forking from SSE [10] and evaluated the feasibility of this technique on a manually instrumented *regression* benchmark. The number of untouched change-annotated labels in the benchmark of Table 3 show the limitation of the SSE approach when trying to analyze reconfigurations that introduce new functionality and modify the interfaces of the POUs. As SSE is driven by concrete inputs from an existing test suite, hitting a change is trivially necessary to exercise it. This also means that important divergences can be missed as it strongly depends on the quality of the initial inputs. There has been work that investigated a full exploration of the four-way fork, not only to a predefined bound, but the experiments have shown that it is intractable in general [14] – it does not scale well. Another downside of the SSE approach in the domain of PLC software lies in the search for additional divergent behaviors. Starting a BSE run from the divergence in the new version leads to the coverage of locations that would have been covered with a more succinct prefix. Due to the cyclic nature, the path prefix of the divergence prevents the coverage of the prior branches – however, it is undecidable in general whether this is redundant or not as it would require a procedure to check before the execution, whether that path is difference-revealing or not.

To conclude, SSE can be used to generate difference-revealing test cases that are suitable for augmentation of the test suite after a reconfiguration. However, it certainly requires further techniques to reduce the amount of generated difference-revealing test cases to benefit the developer during reconfiguration.

Outlook. In future work, we would like to improve our baseline BSE and evaluate more sophisticated merging strategies [16] or the incorporation of incremental solving [12]. While merging may prevent an exponential growth of symbolic execution contexts and can boost the efficiency [11], the reuse of summaries alleviates the analysis by not doing redundant work for paths through the program we have already seen during execution [12]. However, summarization and merging are conflicting techniques as checking whether a summary is applicable or not is based on concrete values, a piece of information we would lose through a merge. It remains unclear how to benefit the most from merging and summarization.

Acknowledgements. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

1. Adiego, B.F., Darvas, D., Viñuela, E.B., Tournier, J.C., Suárez, V.M.G., Blech, J.O.: Modelling and formal verification of timing aspects in large plc programs. IFAC Proc. **47**(3), 3333–3339 (2014). https://doi.org/10.3182/20140824-6-ZA-1003.01279. 19th IFAC World Congress
2. Allen, F.E.: Control flow analysis. In: Northcote, R.S. (ed.) Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, 27–28 July 1970, pp. 1–19. ACM (1970). https://doi.org/10.1145/800028.808479
3. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1-50:39 (2018). https://doi.org/10.1145/3182657
4. Bohlender, D., Simon, H., Friedrich, N., Kowalewski, S., Hauck-Stattelmann, S.: Concolic test generation for PLC programs using coverage metrics. In: Cassandras, C.G., Giua, A., Li, Z. (eds.) 13th International Workshop on Discrete Event Systems, WODES 2016, Xi'an, China, 30 May – 1 June 2016, pp. 432–437. IEEE (2016). https://doi.org/10.1109/WODES.2016.7497884
5. Godefroid, P.: Compositional dynamic test generation. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, 17–19 January 2007, pp. 47–54. ACM (2007). https://doi.org/10.1145/1190216.1190226
6. Grochowski, M., et al.: Formale methoden für rekonfigurierbare cyber-physische systeme in der produktion. at-Automatisierungstechnik **68**(1), 3–14 (2020). https://doi.org/10.1515/auto-2019-0115
7. Gurfinkel, A., Chaki, S.: Boxes: a symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_18

8. International Electrotechnical Commission: IEC 61131-3:2013 Programmable controllers - Part 3: Programming languages. IEC International Standard IEC 61131-3:2013 (2013). https://webstore.iec.ch/publication/4552
9. Jeschke, S., Brecher, C., Song, H., Rawat, D.B. (eds.): Industrial Internet of Things. SSWT, Springer, Cham (2017). https://doi.org/10.1007/978-3-319-42559-7
10. Kuchta, T., Palikareva, H., Cadar, C.: Shadow symbolic execution for testing software patches. ACM Trans. Softw. Eng. Methodol. **27**(3), 10:1-10:32 (2018). https://doi.org/10.1145/3208952
11. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 193–204. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2254064.2254088
12. Lin, Y., Miller, T., Søndergaard, H.: Compositional symbolic execution: Incremental solving revisited. In: Potanin, A., Murphy, G.C., Reeves, S., Dietrich, J. (eds.) 23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, 6–9 December 2016, pp. 273–280. IEEE Computer Society (2016). https://doi.org/10.1109/APSEC.2016.046
13. de Moura, L., 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
14. Noller, Y., Nguyen, H.L., Tang, M., Kehrer, T., Grunske, L.: Complete shadow symbolic execution with java pathfinder. ACM SIGSOFT Softw. Eng. Notes **44**(4), 15–16 (2019). https://doi.org/10.1145/3364452.33644558
15. PLCopen - Technical Committee 5: Safety software, technical specification, part 1: Concepts and function blocks. Technical report, PLCopen (2020). https://plcopen.org/system/files/downloads/plcopen_safety_part_1_version_2.01.pdf
16. Sen, K., Necula, G., Gong, L., Choi, W.: MultiSE: multi-path symbolic execution using value summaries. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 842–853. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2786805.2786830
17. Ulewicz, S., Vogel-Heuser, B.: Industrially applicable system regression test prioritization in production automation. IEEE Trans Autom. Sci. Eng. **15**(4), 1839–1851 (2018). https://doi.org/10.1109/TASE.2018.2810280
18. Weigl, A., Ulbrich, M., Lentzsch, D.: Modular regression verification for reactive systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020, Part II. LNCS, vol. 12477, pp. 25–43. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61470-6_3
19. Weyrich, M., Zeller, A.: Testen von industrie-4.0-systemen - wie vernetzte systeme und industrie 4.0 unser verständnis von systemtest und qualitätssicherung ändern (2016). https://www.ias.uni-stuttgart.de/dokumente/vortraege/2016-01-26_Industrie40_Duesseldorf_v12final.pdf
20. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Trans. Softw. Eng. **17**(7), 703–711 (1991). https://doi.org/10.1109/32.83906
21. Xu, Z., Kim, Y., Kim, M., Cohen, M.B., Rothermel, G.: Directed test suite augmentation: an empirical investigation. Softw. Test. Verif. Reliab. **25**(2), 77–114 (2015). https://doi.org/10.1002/stvr.1562
22. Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM Trans. Softw. Eng. Methodol. **24**(1), 3:1-3:42 (2014). https://doi.org/10.1145/2629536

23. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012). https://doi.org/10.1002/stv.430
24. Zeller, A., Jazdi, N., Weyrich, M.: Functional verification of distributed automation systems. Int. J. Adv. Manufact. Technol. **105**(9), 3991–4004 (2019). https://doi.org/10.1007/s00170-019-03791-2