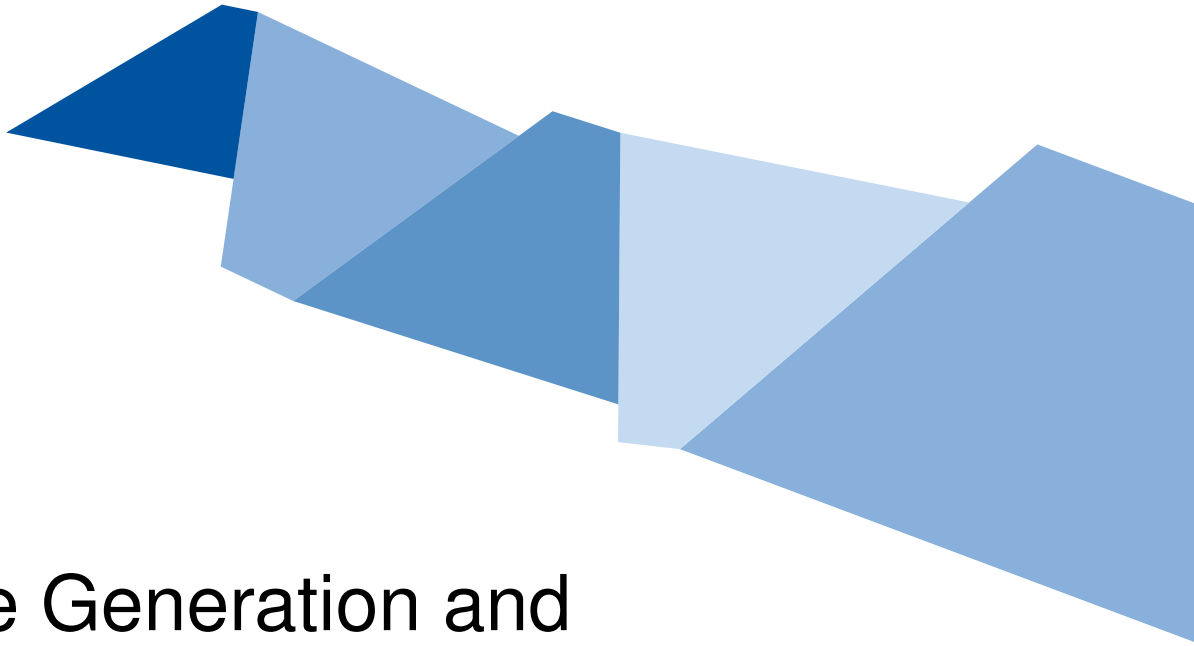


Test Suite Generation and Augmentation for Reconfigurable Industrial Control Software in the Internet of Production

Marco Lutz geb. Grochowski

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2024-09
RWTH Aachen University | Department of Computer Science | September 2024

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Test Suite Generation and Augmentation for Reconfigurable Industrial Control Software in the Internet of Production

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Marco Lutz geb. Grochowski, M. Sc. RWTH
aus Hilden

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. Paula Herber

Tag der mündlichen Prüfung: 10. September 2024

Marco Lutz geb. Grochowski
Lehrstuhl Informatik 11
grochowski@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2024-09

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

With the advent of Industry 4.0 and the digitally networked factory, cyber-physical production systems (CPPSs) are reconfigured frequently along their life cycle to adapt to changing customer requirements or market demands. Such reconfigurations are not limited to the hardware but also affect the software of the programmable logic controllers (PLCs) driving these plants. While verification and testing are two techniques capable of alleviating the risk of introducing errors in production code, it is no longer sufficient to rely only on the results obtained by these methods during the commissioning of the CPPS. Even minor incremental reconfigurations to the PLC's software during the operational phase of the life cycle may introduce regressions that can be quickly overlooked by a developer and therefore need to be reverified.

The goal of this thesis is to provide a "push button" analysis for generating test cases after a static reconfiguration. The generated test cases can be injected and monitored during maintenance or virtual commissioning to observe the impact of reconfiguration on the CPPS by the developer.

In order to reduce redundancy in test suite generation (TSG) after a structural reconfiguration to the PLC software, symbolic summaries of specific parts of the program should be cached and reused to benefit subsequent analysis. While automatic TSG is an established technique used to generate test suites adhering to structural coverage metrics of PLC software, the generated test suite might not anymore be adequate enough with regards to the coverage metric to ensure the absence of regressions. An indispensable part of regression testing (RT) is test suite augmentation (TSA), which guides the TSG toward the reconfigured behavior and increases the chances of deriving difference-revealing test cases which expose behavioral differences between the program and its reconfigured version. The derivation of new test cases is required to uncover potential regressions after a reconfiguration.

To this end, the contributions of this thesis include

- heuristics for the scalability of the existing TSG for PLC software,
- the reuse of symbolic summaries during TSG of reconfigured PLC software,
- and the concept of executing the old and new version of a reconfigured PLC software in one unified program version during TSA.

These contributions are evaluated on selected domain-specific benchmarks of varying difficulty from the PLCopen Safety suite and the Pick and Place Unit (PPU).

Zusammenfassung

Mit dem Aufkommen von Industrie 4.0 und der digital vernetzten Fabrik werden cyber-physische Produktionssysteme (CPPS) während ihres Lebenszyklus häufig neu konfiguriert, um sich an veränderte Kunden- oder Marktbedürfnisse anzupassen. Solche Rekonfigurationen sind nicht nur auf die Hardware beschränkt, sondern betreffen auch die Software der speicherprogrammierbaren Steuerungen (SPSen), die diese Anlagen steuern. Während die Verifizierung und das Testen zwei Techniken sind, die das Risiko von Fehlern des in der Produktion eingesetzten Quellcodes vermindern, reicht es heutzutage nicht mehr aus, sich nur auf die aus der Inbetriebnahme resultierenden Ergebnisse zu verlassen. Selbst kleine inkrementelle Rekonfigurationen an der SPS-Software während der Betriebsphase entlang des Lebenszykluses können Regressionen einführen, die von einem Entwickler schnell übersehen werden können und daher erneut sichergestellt werden müssen.

Das Ziel dieser Arbeit ist es, eine „Push-Button“-Analyse für die Generierung von Testfällen nach einer Rekonfiguration zu entwerfen. Die generierten Testfälle können während der Wartung oder der virtuellen Inbetriebnahme eingespeist und überwacht werden, um die Auswirkungen der Rekonfiguration auf das CPPS durch den Entwickler zu beobachten. Um die Redundanz bei der Generierung von Testsuiten (TSG) nach solchen Rekonfigurationen zu verringern, sollen symbolische Zusammenfassungen von bestimmten Teilen des Programms zwischengespeichert und wiederverwendet werden. Die Testsuite-Erweiterung (TSA) stellt einen unverzichtbaren Teil des Regressionstestens (RT) dar, da die Abwesenheit von Regressionen nach einer Rekonfiguration nicht ausschließlich durch die alte Testsuite gewährleistet werden kann. Die TSA lenkt die TSG in Richtung des rekonfigurierten Verhaltens und erhöht so die Chancen Testfälle abzuleiten, die die Verhaltensunterschiede zwischen beiden Programmversionen aufdecken. Zu diesem Zweck beinhaltet der Beitrag dieser Arbeit

- Heuristiken für die Skalierbarkeit der bestehenden TSG für SPS-Software,
- die Wiederverwendung von symbolischen Zusammenfassungen während der TSG von rekonfigurierter SPS-Software,
- und das Konzept der Ausführung der alten und der neuen Version einer rekonfigurierten SPS-Software in einer Programmversion für die TSA.

Diese Ansätze werden anhand ausgewählter domänenspezifischer Benchmarks mit unterschiedlichem Schwierigkeitsgrad aus der PLCopen Safety Suite und der Pick and Place Unit (PPU) bewertet.

Contents

1 Introduction

Since the advent of the Industrial Internet of Things (IIoT), conventional manufacturing has experienced a paradigm shift toward cyber manufacturing. This shift is driven by the growing demands for individual products and emerging information and communications technologies (ICT) [Jes & Bre+ 17], resulting in the fourth industrial revolution known as Industry 4.0 [Bun 16].

As conventional production systems' life cycles and value chains are too rigid regarding the increasingly demanded agility in industrial automation [Gro & Sim+ 20], the trend goes toward networking and distributed computing [Bor & Tre+ 21]. While these technologies are well understood, the currently applied concepts must be adapted to meet future production requirements.

Several challenges for formal and semi-formal methods arise with service-oriented architectures (SOAs) as enablers and implementation means for multi-agent systems (MASs). The increasing modularization and distributed control structures result in heterogeneity regarding ad-hoc networking and a high degree of reconfigurability, leading to emergent behavior and potential regressions after the production system is put into operation.

Therefore, the quality assurance of logic control software during the life cycle of a cyber-physical production system (CPPS) requires adaptations in the testing and verification processes known from traditional software engineering. With the use of Digital Shadows (DSs) and the insights gained from production, reconfigurations made to the logic control software need to be adequately tested and verified to ensure safe changes to the production process. The central challenge lies in the aggregation, abstraction, and analysis of heterogeneous data [Bra & Dal+ 22], which the Internet of Production (IoP) tries to master.

1.1 Internet of Production

The IoP builds on the ideas of the IIoT and advocates the vision of enabling real cross-domain and inter-company collaboration by providing semantically adequate and context-aware data from production [Pen & Gle+ 19]. Figure 1.1 shows the architecture and infrastructure of the IoP in the background and a high level abstraction in the foreground. During the life cycle of the production process, data emitted by CPPSs are collected and analyzed. The insights gained from the emitted data are turned into data that controls the process and forms the basis for changes, which create new services and flexible value chains [Gro & Kow+ 19]. Changes translated into reconfigurations of the underlying logic control

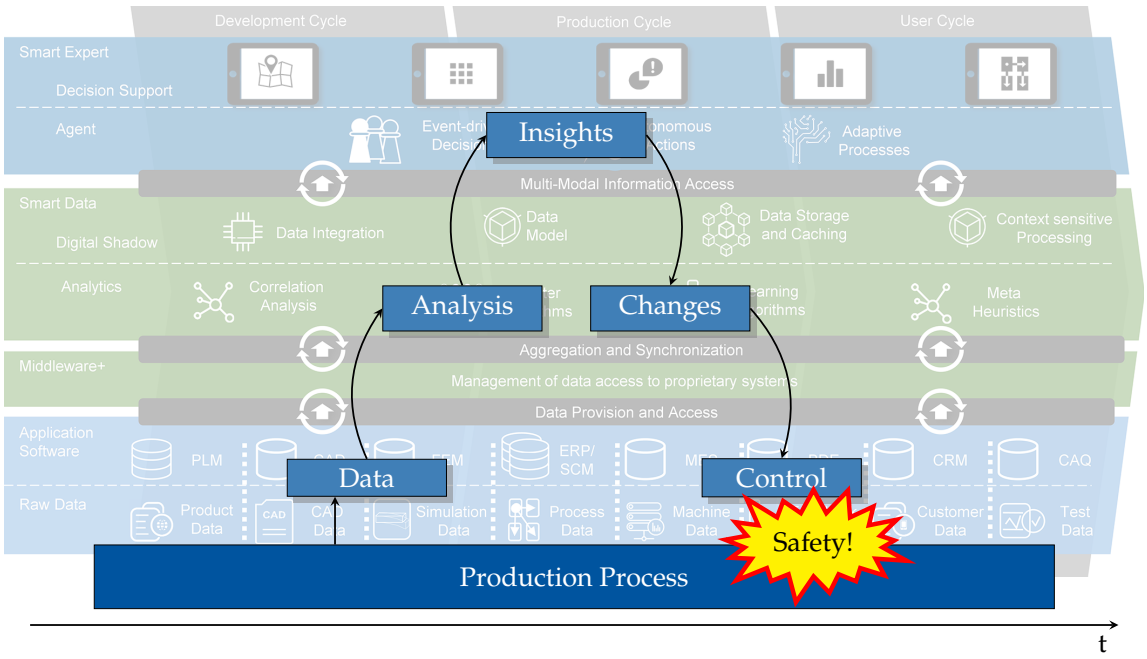


Figure 1.1: The architecture and infrastructure of the IoP [Bre & Klo+ 17].
Figure adapted from Bild 1 in [San & Xi+ 21].

software of the CPPS can result in safety-critical modifications requiring additional safeguarding.

1.1.1 Vision, Objective, and Impact

The vision, objective, and impact of the IoP concerning the functional perspective for the realization of the emphasized process depicted in Figure 1.1 are listed next.

Vision:

- Integration of machine learning and AI-based models from engineering
- Creation of IoP-specific "data to knowledge pipelines" that transform massive data into insights
- Providing meaningful, actionable knowledge to decision-makers

Objective:

- Visualizations, decision support, and human-centered interfaces
- Enable cross-learning and transfer
- Data-to-knowledge pipelines

Impact:

- Methods for data-driven insights and back-coupling to transform insights into actions – usable by human or machine
- Smart Decision Support for the human-in-the-loop
- Toolbox for Digital Shadows, Data-To-Knowledge Pipelines, validated, self-adaptive production systems

The key challenges for the IoP lie in building algorithms that combine machine learning with model-based analysis and control and develop human-centered interfaces to facilitate decision support.

1.1.2 Digital Shadow

A key component in realizing the vision, objective, and impact of the IoP is the digital shadow (DS).

> **Definition 1.1: Digital Shadow [Bib & Dal+ 20; Bec & Bib+ 21]**
>
> A digital shadow (DS) is a set of temporal data traces and/or their aggregation and abstraction collected concerning a system for a specific purpose, i.e., context-based with respect to the original system.

The DS comprises task-specific data of production processes allowing the reconstruction of the entire life cycle of an industrial asset and serves as the primary technique for data aggregation and refinement within the IoP [Bre & Buc+ 19]. The DS, therefore, abstracts from the underlying production process and serves as a virtual representation of the current state of the logic control software.

This gives rise to the use of event-driven architectures in which messages are exchanged as soon as a state change occurs [Bre & Buc+ 19] and is further discussed in Section 1.2.1. The use of the DS for the contribution of this thesis lies in the aggregation of data during the whole analysis life cycle of test suite generation (TSG) and test suite augmentation (TSA) of a reconfigured programmable logic controller (PLC) program. In particular, the specific purpose mentioned in Definition 1.1 refers to the artifacts resulting from the process of TSG and TSA, and hence the generated test cases and function block (FB) summaries form the DSs of testing. This disruptive concept of the IoP results in transformable production systems.

1.2 Transformable Production Systems

Transformability is one of the primary enablers of coping with changing intrinsic and extrinsic demands. It is a required property to guarantee competitiveness among companies and comprises the flexibility and mechanical reconfigurability of

a CPPS [Jes & Bre+ 17]. Flexibility describes the property of a CPPS to adapt quickly and with little effort, within the limits of a given range, to changed conditions and, together with reconfigurability, enables the CPPS to adapt to changes that were unknown during planning [Rei & Kre+ 08; Wie & Rei+ 14]. Figure 1.2 shows the

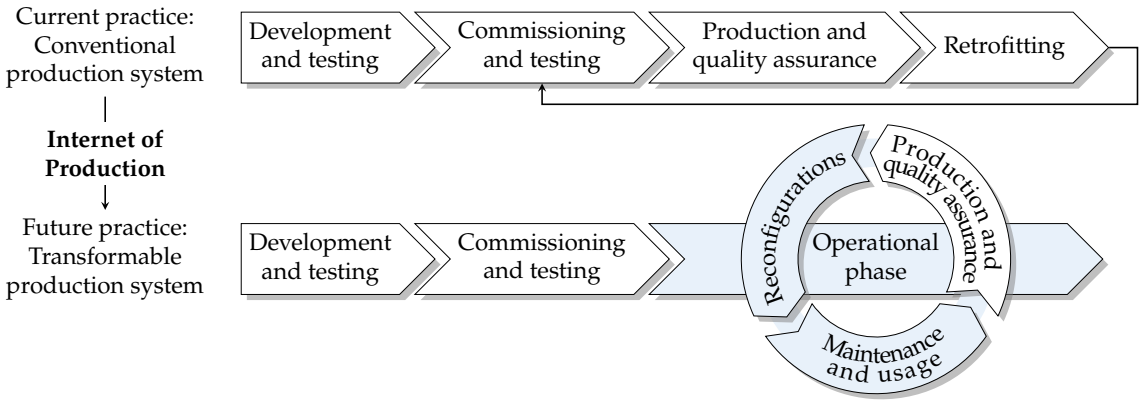


Figure 1.2: Juxtaposition of the production system's life cycle and value chain. *Figure adapted from Fig. 2 in* [Zel & Wey 15].

juxtaposition of conventional and transformable production systems' life cycle and value chain. This shift from conventional to transformable production systems comes with a series of challenges that the IoP tries to master.

In general, the overall complexity of the CPPS increases [Wie & Rei+ 14] as the IoP blurs the distinction between the development and operational phases of the production system.

While transformability is a crucial enabler for quickly adapting to changing market requirements, it also increases the complexity due to the reconfigurability and emergent behavior [Wie & Rei+ 14; Ste & Hei 17]. This leads to shorter product cycles and an increase in the number of software variants. This highly iterative development and agile manufacturing process in which the requirements are subject to continuous changes take their toll on the safety of the CPPS. To manage the increasing complexity, efficient and lightweight techniques that bridge the gap between verification and testing are required.

The following section explains the heterogeneous environments and the distributed control of a CPPS.

1.2.1 Service-oriented Architecture

A SOA, in combination with the concept of the DS, suits the demanding requirements of reconfigurable CPPS [Bre & Buc+ 19]. Figure 1.3 shows the composition of services to processes and the distinction between *orchestration* and *choreography* [Pel 03; Rie 12]. A centralized service coordinates an orchestration, e.g., "Service A" or "Service D" in Figure 1.3, which implements the business logic or the production process workflow. This centralized service describes all necessary information

for the aggregation of multiple services, their interfaces, their dependencies with regard to the flow of control, and their exchanged messages [Rie 12].

In contrast, the *choreography* does not follow some centralized coordination but instead consists of the interplay between autonomous services in which their cooperation may fulfill cross-organizational processes. It specifies the communication protocol and works on the *observable behavior* of the participating processes, commonly specified as behavioral interfaces [Rie 12]. The depicted SOA in Figure 1.3

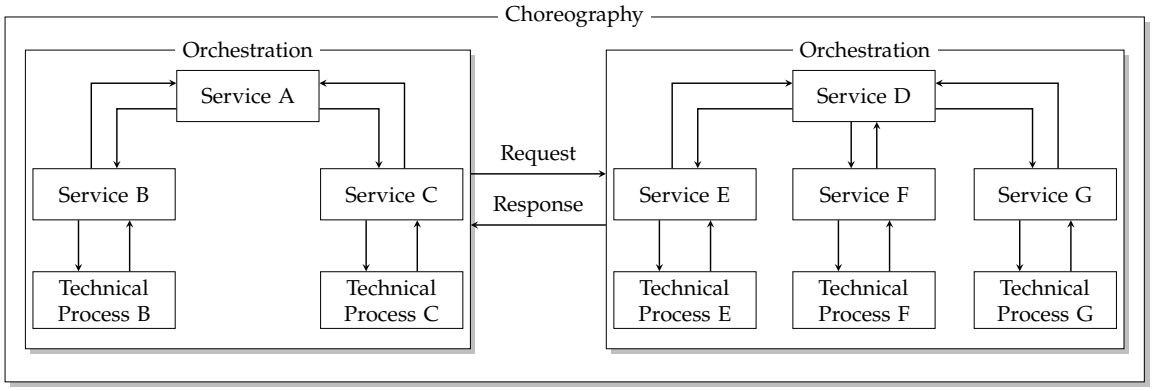


Figure 1.3: Orchestration and choreography in a SOA.
Figure adapted from Figure 1 in [Pel 03].

is therefore subdivided by the orchestration and choreography into two views on the system, a *local* and a *global view*, respectively [Rie 12]. The application of formal methods on the global view can thus only assure the interoperability between the processes in this choreography. As opposed to this, the application of formal methods on the local view allows for verification of the behavior of the services due to the centralized and coordinating service and the knowledge about the actual business logic [Rie 12].

This thesis focuses on the safeguarding of the local view. To further delimit the contribution, Figure 1.4 refines the local view by further splitting it up into an *exterior* and an *inner view* [Zel & Wey 18]. The "exterior" view communicates to other components via its interface and abstracts from the technical process, whereas the "inner" view focuses on communication with the underlying technical process and is responsible for driving the technical process to behave in the desired manner. Typically, reactive systems are used to control the technical process.

In the IoP, the reactive nature of the logic control software is lifted to an event-based system with the help of the DS and exposed as a service [Gro & Kow$^+$ 19]. As both components, i.e., services, and their assemblies, i.e., composed services, can be subject to changes in an agile environment, the following section talks about reconfigurations that can occur.

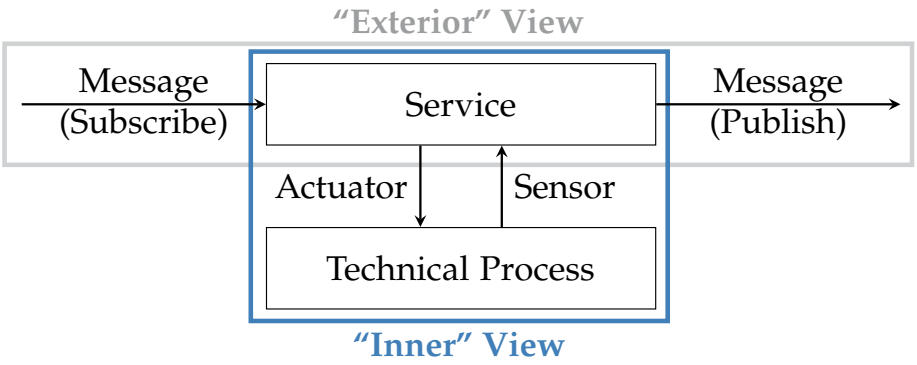


Figure 1.4: Generalized views on a service with exemplary communication technologies and its interaction with a technical process.
Figure adapted from Fig. 1. a) in [Zel & Wey 18].

1.2.2 Reconfigurations of Production Systems

Reconfigurations allow the CPPS to be able to change quickly and cost-effectively in terms of their capacities, their functional contents, and the technologies they make available to meet the demands of today's markets [Ste & Hei 17]. It enables quick scale-up and change management using modular structures with matching interfaces [Ste & Hei 17] and pushes the boundaries of the CPPS's flexibility corridors. In order to unify the terminology of *evolution* [Bec & Mun+ 19], *changes*, and *adaptations* [Vog & Rös+ 16] found in literature, the definition of *reconfiguration* is used synonymously throughout this thesis.

> **Definition 1.2: Reconfiguration [Mat 10]**
>
> *"A reconfiguration represents the technical view of the process of modifying an already developed and operationally deployed system to adapt it to new requirements, extend functionality, eliminate errors, or improve quality characteristics."*

Due to this increasing agility in the development process, the shorter life cycles, and the changing customer and legislator requirements, the software maintenance process must account for reassurance of the CPPS's behavior after a reconfiguration [Gro & Sim+ 20]. A reconfiguration can affect hardware as well as software, and a classification for the types of reconfiguration in accordance with the definition of [Mat 10] is illustrated in Figure 1.5.

In this thesis, reconfigurations affecting the logic control software are analyzed. These reconfigurations are often initiated by changes to the hardware components of the CPPS and form the basis for software evolution. Reconfigurations of a CPPS can either be structural (architecture-based), functional (service-related), or non-functional (quality-related) [Mat 10] and affect either the "exterior" or "inner" view. In the subsequent sections, these types of reconfigurations are explained

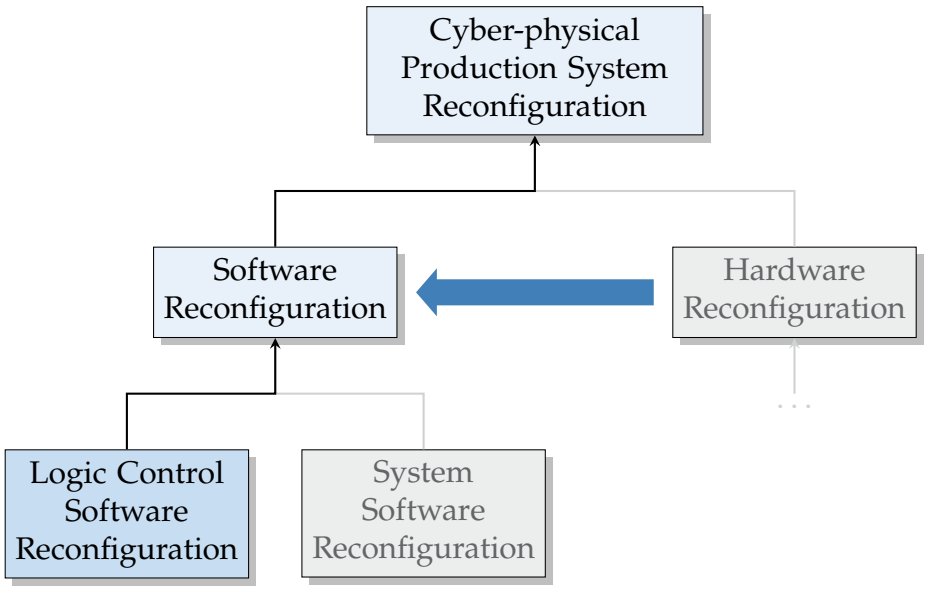


Figure 1.5: Application areas for reconfigurations of CPPS.
Figure adapted from Abbildung 5.1 in [Mat 10].

with the help of the two perspectives illustrated in Figure 1.4.

Reconfigurations of the Exterior View

Structural reconfigurations of the exterior view are, for instance, changes in the services' interfaces or changes in the dependencies between the services or the architecture. These reconfigurations do not impact the behavior of the inner view, as the control logic is encapsulated within a service. Figure 1.6 summarizes the effect of modifications, deletions, and additions on the level of services. Modifications to the service interface do not impact the implementation that controls the technical process.

However, the converse is not true. Typically, behavioral reconfigurations of the "exterior" view are achieved by either changing the interfaces of the "exterior" view or changing the behavior, i.e., the implementation of the "inner" view. The deletion of an existing service, such as Service C, or the addition of a new service, such as Service D, depicted in Figure 1.6, only affects the business logic of Service A and its interoperability but does not affect any non-directly connected services.

In general, structural reconfigurations of the "exterior" view do not impact the behavior of the "inner" view, and verification boils down to proving interoperability using behavioral information. More interesting are reconfigurations impacting the exterior view that stem from reconfigurations of the inner view and are discussed in the following.

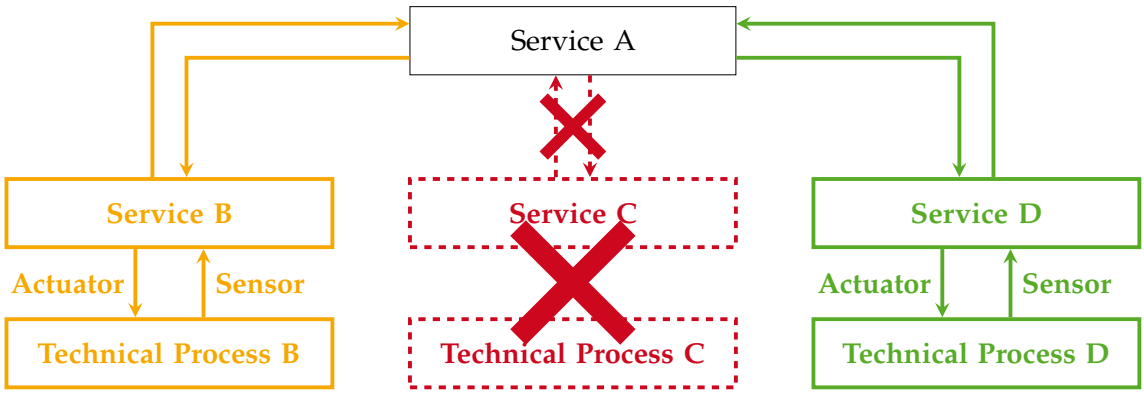


Figure 1.6: Reconfiguration of a SOA after modifications to an existing service, deletion of a service and/or connections, and addition of a new construction to the production system, which requires an implementation of an additional service or addition of connections.

Reconfigurations of the Inner View

Typically, with regard to logic control software, reconfigurations originate from reconfigurations occurring to the technical process, as depicted in Figure 1.5. Reconfigurations to the technical process can be categorized as the addition, modification, and removal of new or existing hardware components [Vog & Fol+ 14].

1. Addition of a new hardware component:
   - Implementation of a new function block
   - Adaptation of an existing function block with regard to its interface and implementation
2. Modification of the behavior of the context (hardware/mechanical):
   - Modification of the internal structure of the function block
   - Optimization of the behavior of the function block
3. Removal of a hardware component:
   - Reduction of the interface and internal structure of the function block

These hardware reconfigurations often induce reconfigurations in the software and are illustrated in Figures 1.7 to 1.9. For example, when a new hardware component, such as a sensor or actuator, is added, the functionality must be likewise represented in software.

Therefore, a new FB may be implemented to account for the new functionality, as depicted in Figure 1.7, or the adaptation of an existing FB with regard to its interfaces and implementation. A modification of the behavior of the technical

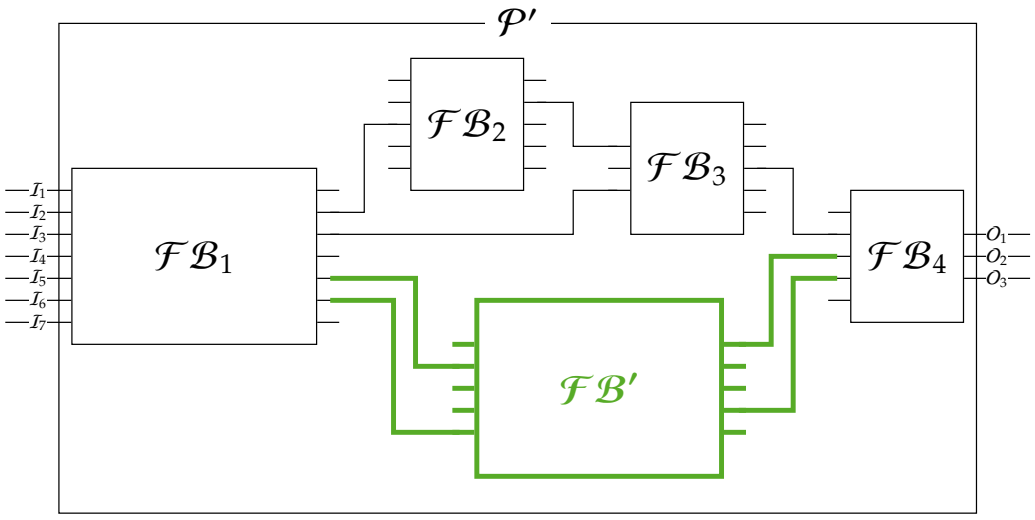


Figure 1.7: Reconfiguration of a program after adding a new construction to the CPPS, which requires an implementation of an additional FB $\mathcal{FB}'$.
Figure adapted from Abbildung 4 (1) in [Vog & Fol+ 14].

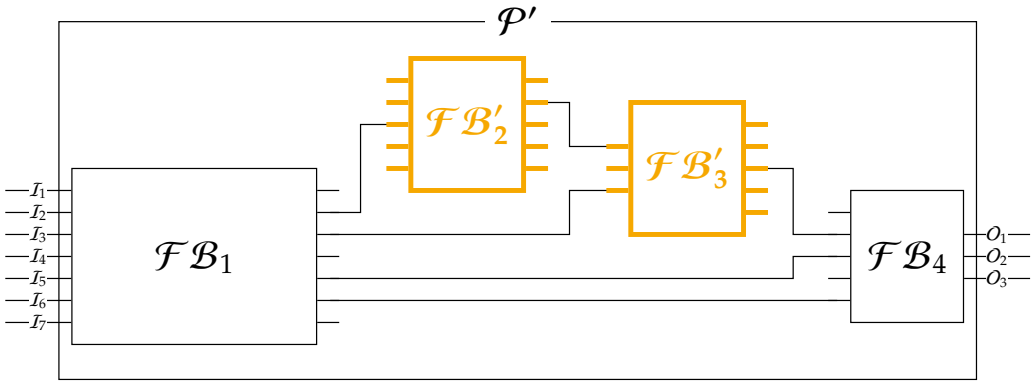


Figure 1.8: Reconfiguration of a program by modifying the behavior of the context (mechanical) to the production system, which requires the adaptation of the internal structure of the program. The interface may stay the same, e.g., when the order of two processing steps is changed.
Figure adapted from Abbildung 4 (3) in [Vog & Fol+ 14].

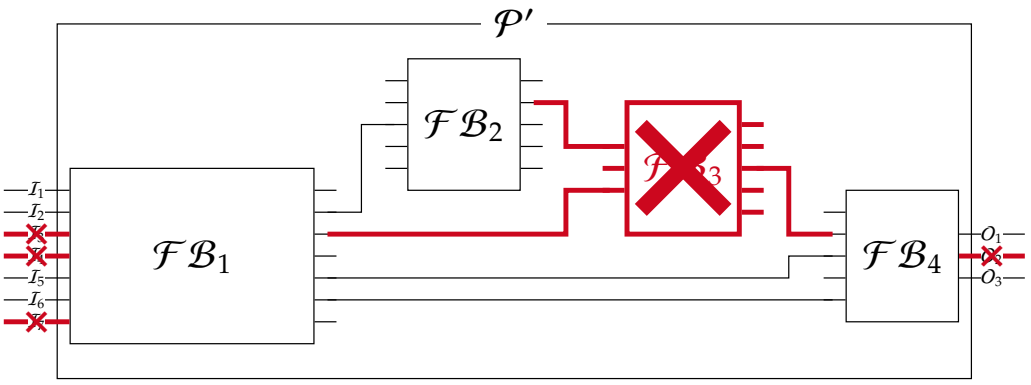


Figure 1.9: Removal of a hardware component can lead to the removal of FBs and modifications to the interface.
Figure adapted from Abbildung 4 (5) in [Vog & Fol+ 14].

process may lead to a change in the structure of the underlying software. Figure 1.8 reflects the reconfiguration of the technical process in software. Last, removing a hardware component can reduce the interface and the internal structure of the FB, as depicted in Figure 1.9.

Depending on the intended hardware changes, complex software reconfigurations can arise. However, they can be reduced to a combination of small incremental reconfigurations, as illustrated by the types of reconfigurations in Figures 1.7 to 1.9. In the next section, the impact of such planned and unplanned reconfigurations is categorized and integrated into the software maintenance process.

1.3 Software Maintenance Process

The quality assurance of logic control software during the life cycle of a CPPS requires adaptations in the testing and verification processes known from traditional software engineering. With the use of the DS and the insights gained from production, reconfigurations made to the control software need to be adequately tested and verified to ensure safe modifications to the production process. In the context of software maintenance, the data of the DS does not only encompass the data passed in communication between services but also artifacts resulting from the maintenance process. CPPSs are reconfigured frequently during their life cycle, and hence it must be ensured that a revision does not introduce any regressions while achieving the intended effect [Bec & Mun+ 19].

Figure 1.10 gives an overview of the software maintenance process. A reconfiguration request starts the software maintenance process. Reasons for reconfigurations consist of minimizing efforts, defects, or cost. It is not relevant whether the reconfiguration request is intrinsic or extrinsic, i.e., either triggered internally by the developing company or induced externally by another company or the market.

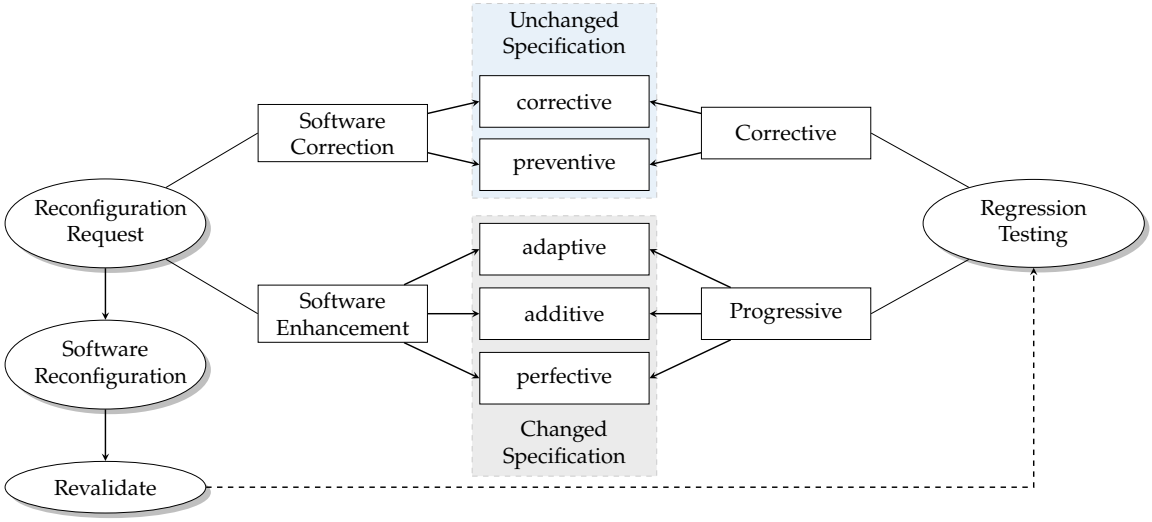


Figure 1.10: Overview of the software maintenance process and integration of regression testing as a method for revalidation.
Figure adapted from Fig. 19 in [Gal & Lyl 91].

More important is the distinction between the type of reconfiguration request, either software correction or software enhancement. Depending on the type of reconfiguration, the specification is either changed or unchanged. Changes to the specification are classified as adaptive, additive, or perfective reconfigurations [ISO 22].

The class of software corrections can either be corrective, i.e., a bug was fixed, or preventive, i.e., to correct faults in the software before they occur in the system [ISO 22]. After the reconfiguration request has been stated, the action follows. The software is reconfigured, and the effect of the reconfiguration needs to be revalidated to confirm no unintended changes have occurred. An in general undecidable but very sought-after question (cf. [Pod & Cla 90]) during software maintenance is how a change in the semantics of a program statement affects the execution behavior of other statements in the program.

The goal is to find reasonable approximate solutions for the following two problems: (1) determining the set of affected components and (2) determining the set of tests that exercise these components. A technique that tries to answer this question is regression testing, which can either be corrective or progressive and is further discussed in the next section.

1.3.1 Regression Testing

The naive approach, after a reconfiguration, uses an existing test suite and reruns it on the reconfigured PLC program. While this works in theory, it tends to yield two practical problems:

1. If the test suite is too large, executing all test cases might not be feasible. Therefore, some form of prioritization or minimization has to be performed [Yoo & Har 12].

2. The existing test suite might not test the changed behavior and therefore requires TSA [Xu & Kim+ 15]

Regression testing typically consists of four major problems: (1) *test suite classification*, (2) *test suite execution*, (3) *test suite coverage identification*, and (4) *test suite maintenance* [Leu & Whi 89; Rot & Har 96]. The problem of test suite classification deals with the selection of test cases $T' \subseteq T$ such that specific criteria such as required time to test are minimized, and other criteria like coverage are maximized. Figure 1.11 gives an overview of the regression testing pipeline and the software

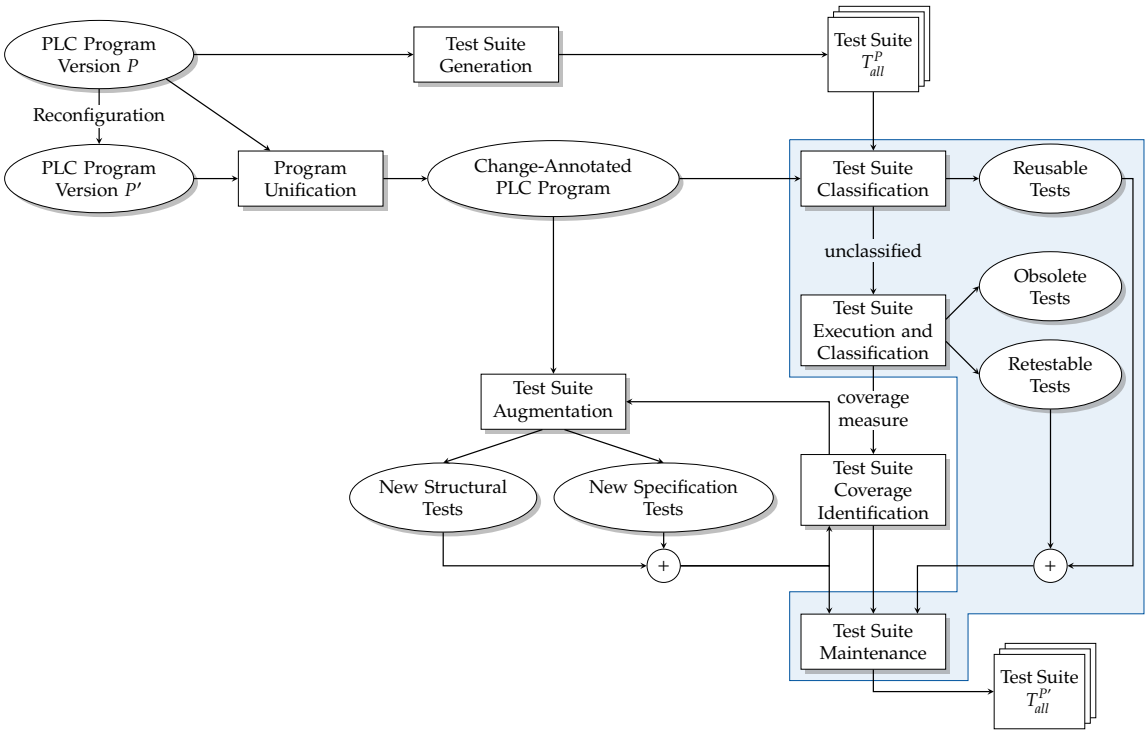


Figure 1.11: Overview of the regression testing pipeline and maintenance. *Figure adapted from Figure 3 in* [Leu & Whi 89].

maintenance process and is referenced throughout this section. Figure 1.12 illustrates the three main techniques: test (1) *selection*, (2) *prioritization*, and (3) *minimization* [Yoo & Har 12].

Test case selection aims to remove all test cases that are not part of the retest set. The resulting set of test cases can still be "too big" to test because it would, for example, require more time than available after a reconfiguration. Test case prioritization can use the execution history and is a widely utilized technique for regression testing [Eng & Run+ 10] to prioritize new-structural and new-specification

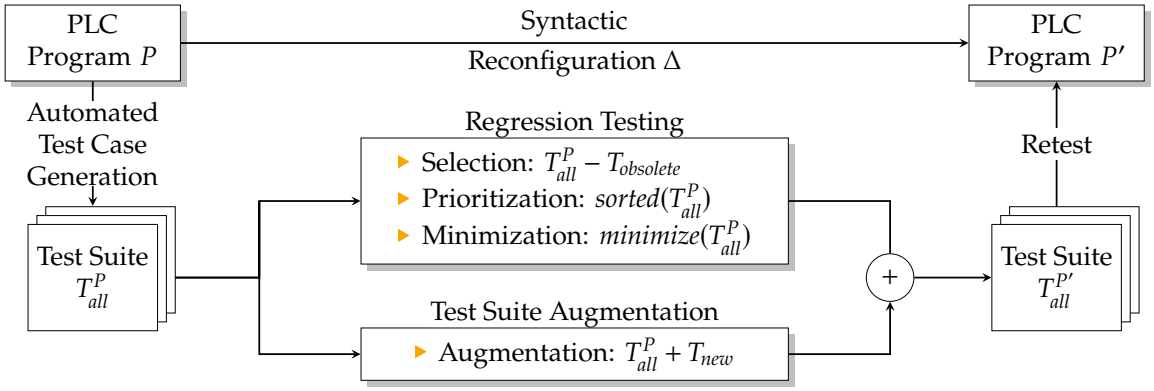


Figure 1.12: Application of regression testing techniques and test suite augmentation after a syntactic reconfiguration.

test cases. Last, test suite minimization tries to reduce the number of test cases required to cover as many faults as possible and corresponds to the minimal set cover problem, which is NP-complete [Yoo & Har 12] and, therefore, rarely used. There are, however, some heuristics but most regression testing pipelines rely on selection techniques that are modification-aware [Rot & Har 96]. A complementary technique to regression testing poses TSA [Xu & Kim+ 15].

The technique presented in Chapter 4 can be categorized as some form of regression algorithm, as already done work tries to be lifted to the reconfigured program version. The test suite execution problem tries to establish the correctness of the reconfigured program version P' with respect to the T'. The test suite coverage identification problem is the subject of Chapter 5. The goal is to identify whether the previous test suite T is still sufficient to cover the reconfigured program version P' and, if necessary, create a set of new functional or structural tests. Last but not least, the test suite maintenance problem deals with creating a new test suite and test history for the reconfigured program P' by consolidating the results from the prior steps [Leu & Whi 89].

Test cases can be classified into the following three categories: (1) *reusable*, (2) *retestable*, and (3) *obsolete* test cases [Leu & Whi 89]. Reusable test cases are not modification-traversing as they execute the parts of the program that remain unchanged between the two versions. They should produce the same result in both program versions and need not be rerun. During test suite classification, reusable tests can be derived statically by analyzing the control-flow graphs (CFGs) of the change-annotated program (CAP), as depicted in Figure 1.11. Retestable test cases, however, are modification-traversing and must be rerun as they execute the parts of the program that have been changed. This class of test cases is especially interesting, as they may be difference-revealing. A test case is said to be difference-revealing if executed on both program versions, the base version P and the reconfigured program P', produce different outputs. Both the reusable and retestable test cases belong to the class of non-obsolete test cases, as depicted

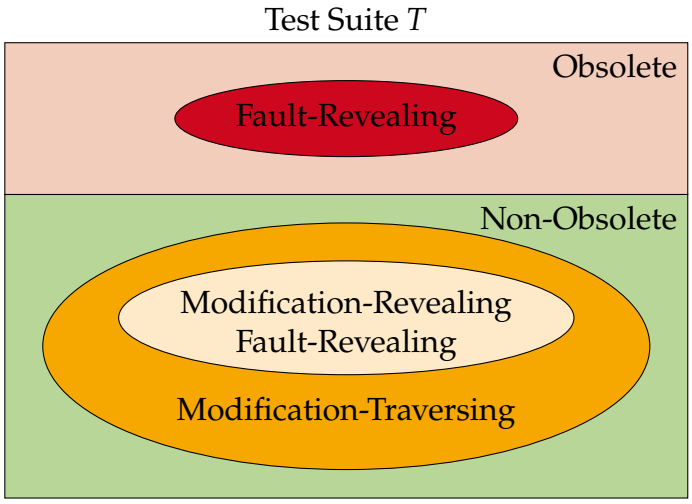


Figure 1.13: Relationship between test classes.
Figure adapted from Fig. 1 in [Rot & Har 96].

in Figure 1.13. Last, the class of obsolete test cases includes test cases that can no longer be used because they were either specified over a different input/output relation or do not contribute to the structural coverage [Leu & Whi 89].

As shown in Figure 1.11, the test suite must be executed to determine whether a test case is retestable or obsolete. For the sake of completeness, the test classes, their relation to the software maintenance process from Figure 1.10, and their target construct are illustrated in Table 1.1. The generation of new-structural and

Table 1.1: Classification of test cases.
Table adapted from Table 2 in [Leu & Whi 89].

Test Class	**Specification**	**Target Construct**	**Test Type**
Reusable	Unchanged	Unchanged	Structural, Specification
Retestable	Unchanged	Changed	Structural, Specification
Obsolete	Unchanged	Changed	Structural
	Changed	Unchanged/Changed	Specification
New-Structural	Unchanged/Changed	New	Structural
New-Specification	Changed	New	Specification

new-specification test cases are part of TSA and hence not discussed here. The test suite coverage identification depicted in Figure 1.11 is discussed and explained in Chapter 5. Last but not least, after the classification of the old test suite and the generation of new test cases, the process of test suite maintenance (TSM) is performed.

Figure 1.14 gives an overview of how the newly derived test plan relates to the old test plan under consideration for corrective and progressive changes, as

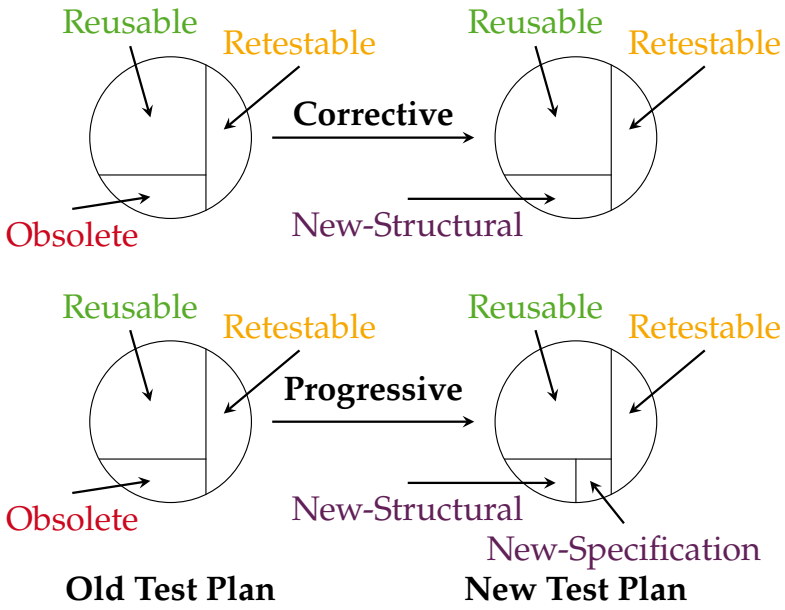


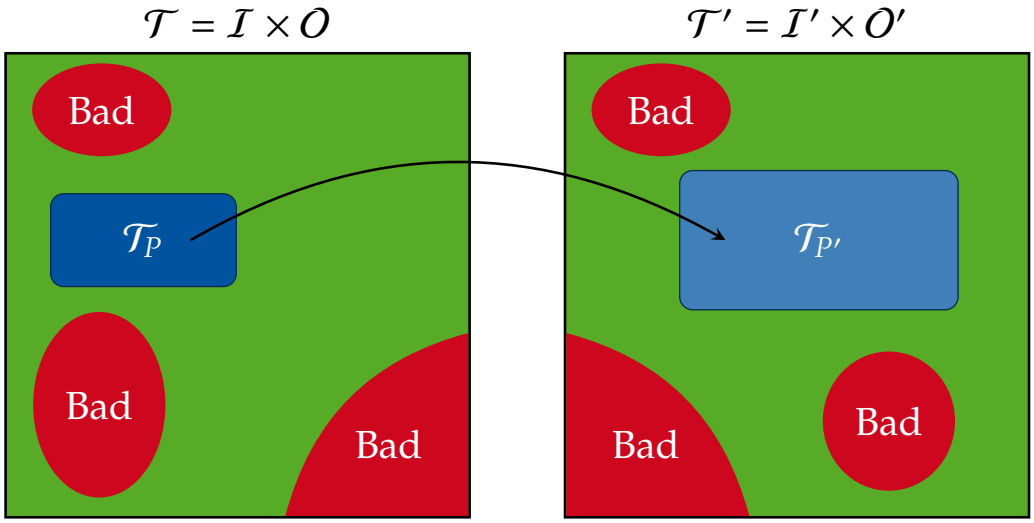
Figure 1.14: Evolution of the test plan during TSM.
Figure adapted from Figure 1 in [Leu & Whi 89].

illustrated in Figure 1.10. In both cases, the TSM amounts to updating the test suite with the newly generated test cases while removing obsolete ones yielding a test suite suitable for testing the reconfigured program version.

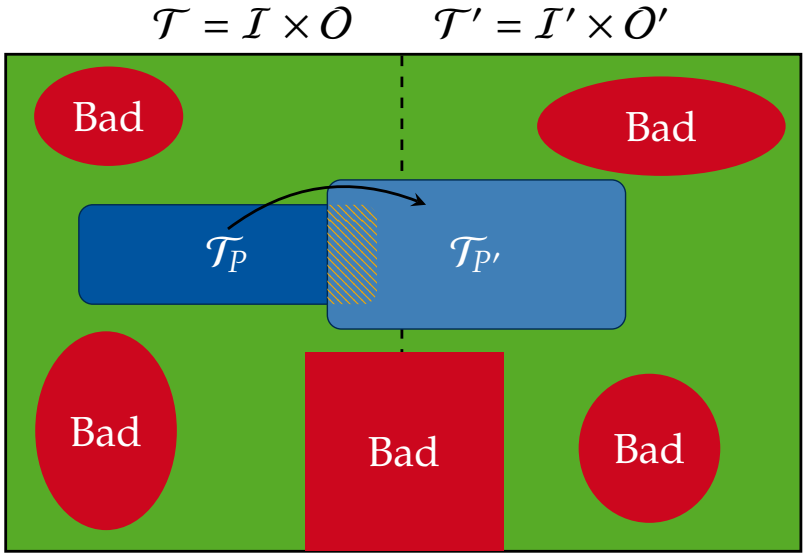
1.3.2 Implications of Reconfigurations on the Trace Semantics

The implications of reconfigurations on the trace semantics are illustrated in Figure 1.15. They either lead to a change of the input/output relation or no change of the input/output relation. The behavior of the base version P and of the reconfigured program P' are expressed by their trace semantics, where $\mathcal{T}$ represents the set of all traces of possible combinations of inputs and outputs, and $\mathcal{T}_P$ the possible traces of P [Cha & Ulb$^+$ 19]. The green area depicts the set of all possible traces that do not violate requirements, whereas the red areas depict traces that exhibit bad behavior in some form or another.

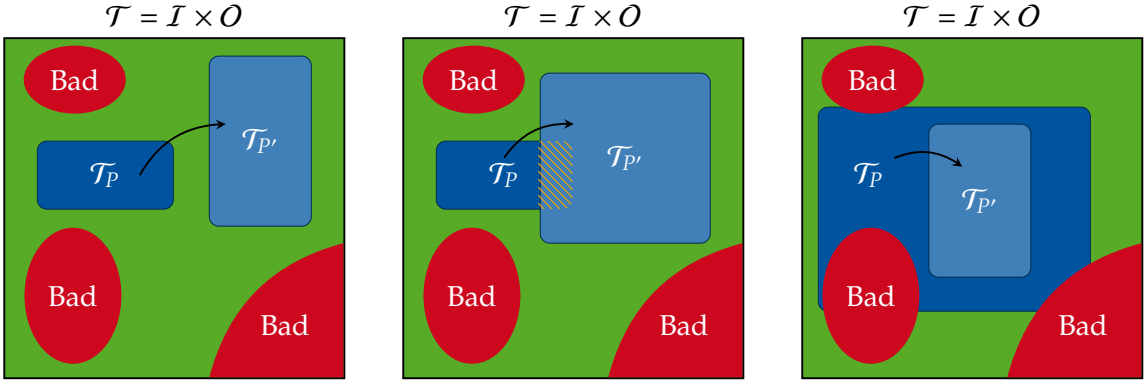
Figure 1.15a shows a complete change of the input/output relation denoted by the non-overlapping domains. Both a complete and a partial change of the input/output relation require new test cases, as all prior generated test cases are obsolete. In these scenarios, techniques other than TSA are required to establish transferrable knowledge. For example, regression verification can establish an equivalence relation between the two versions even though their interfaces have changed [Cha & Ulb$^+$ 19]. However, this change must be captured explicitly by specifying the semantic difference in a logical formula as input for the regression



(a) Complete Change of the Input/Output-Relation.



(b) Partial Change of the Input/Output Relation.



(c) No Change of the Input/Output Relation.

Figure 1.15: Implications of reconfigurations on the trace semantics of PLC programs.
Figure adapted from Fig. 4 in [Cha & Ulb+ 19].

verification algorithm.

The contributions of this thesis consider no changes in the input/output relation of the reconfigured PLC program and hence deal with the class of implications depicted in Figure 1.15c. This is a typical scenario where code is refactored, the technical process is reconfigured without affecting the controller's interface, or additional behavior is implemented [Cha & Ulb+ 19].

1.4 Contribution

The contribution of this thesis is aligned in the following with regard to the vision, objective, and impact of the IoP, as presented in Section 1.1.1. The overarching goal is to ensure safety for rapidly changing CPPS in the IoP in an efficient way. The central research question concerns how traditional software verification and testing methods can apply to emerging technologies in the context of the IoP while fulfilling standard industrial requirements.

The *vision* is to build a data-to-knowledge pipeline to provide meaningful, actionable knowledge, where the data represents the source code of a PLC program and the actionable knowledge of whether a PLC program is safe or unsafe after a reconfiguration.

The *objective* is to provide decision support, i.e., to provide insights into the quality of the development process by evaluating the adequacy of a test suite for structural coverage of a reconfigured PLC program.

The *impact* are the formal methods that provide data-driven insights into the impact of the reconfiguration on the PLC software and a toolbox for DS of testing in the form of a program analysis framework (PAF) as a "push-button" analysis to generate test cases and augment test suites after a reconfiguration.

1.4.1 Publications

The main contributions found in Chapter 4 and Chapter 5 have been published in prior work. This section presents the publications relevant to this thesis and the corresponding contributions.

Relevant Publications

[Gro & Sim+ 20] M. Grochowski, H. Simon, D. Bohlender, S. Kowalewski, A. Löcklin, T. Müller, N. Jazdi, A. Zeller, and M. Weyrich, "Formale Methoden für rekonfigurierbare cyber-physische Systeme in der Produktion", *Autom.*, vol. 68, no. 1, pp. 3–14, 2020. DOI: 10.1515/auto-2019-0115. [Online]. Available: https://doi.org/10.1515/auto-2019-0115.

[Gro & Völ+ 22a] M. Grochowski, M. Völker, and S. Kowalewski, "Automatic Test Suite Generation for PLC Software in the Internet of Production", in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022, pp. 1–8. DOI: 10.1109/ETFA52439.2022.9921726.

[Gro & Völ+ 22b] M. Grochowski, M. Völker, and S. Kowalewski, "Test Suite Augmentation for Reconfigurable PLC Software in the Internet of Production", in *Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings*, J. F. Groote and M. Huisman, Eds., ser. Lecture Notes in Computer Science, vol. 13487, Springer, 2022, pp. 137–154. DOI: 10.1007/978-3-031-15008-1_10. [Online]. Available: https://doi.org/10.1007/978-3-031-15008-1%5C_10.

Further Publications

[Gro & Kow+ 19] M. Grochowski, S. Kowalewski, M. Buchsbaum, and C. Brecher, "Applying Runtime Monitoring to the Industrial Internet of Things", in *24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2019, Zaragoza, Spain, September 10-13, 2019*, IEEE, 2019, pp. 348–355. DOI: 10.1109/ETFA.2019.8869447. [Online]. Available: https://doi.org/10.1109/ETFA.2019.8869447.

Bibliographic Notes and Contributions

The core idea of the TSG algorithm presented in Chapter 4 is based on prior work by Dimitri Bohlender and Hendrik Simon [Sim & Fri+ 15; Boh & Sim+ 16]. While a framework for the analysis of PLC software already existed in the form of the tool ARCADE.PLC [Bia & Bra+ 12], the compilation and translation pipeline from Structured Text (ST) to the intermediate representation (IR) of Sebastian Biallas has been reimplemented by me. All other contributions of this thesis have been implemented in a standalone project by myself. The used benchmarks of the PLCopen Safety suite were implemented by Sebastian Biallas and Hendrik Simon, while I extended it by user-defined program examples.

For compliance with the doctoral regulations, I delimit my contributions from the contributions of my co-authors of the relevant publications. The overview paper [Gro & Sim+ 20] was a joint collaboration with researchers from the University of Stuttgart. Dimitri Bohlender and I worked in close collaboration writing about the work of Hendrik Simon and added our own perspectives in the respective sections 2, 4, and 5.

The paper [Gro & Völ+ 22a] formed the basis of Chapter 4 and benefited from the discussions with Marcus Völker and the valuable feedback provided by Stefan

Kowalewski. The core idea and the underlying implementation of the methodologies were researched and implemented by me.

Chapter 5 is based on [Gro & Völ+ 22b], to which I contributed all original concepts and methodologies. After identifying the need for TSA after a reconfiguration, I supervised Johannes Neuhaus in developing a prototypical implementation in his Master's thesis. While his work gave a good first impression of how TSA could be applied, several improvements and additional features had to be implemented by me. The publication [Gro & Völ+ 22b] thus represents my own work incorporating the discussions and feedback of Marcus Völker and Stefan Kowalewski.

1.4.2 Limitations and Assumptions

This dissertation does not deal with the whole safety life cycle of a CPPS but instead places particular emphasis on analyzing the impact of reconfigurations. While formal verification methods are suitable for safeguarding safety-critical functions, they require a high modeling effort [Gro & Sim+ 20]. This may inhibit the use of formal verification methods in practice, giving rise to more lightweight techniques such as regression testing and TSA. A necessary prerequisite for the algorithms presented in this thesis is the existence of a syntactically change-annotated PLC program given as input to the PAF. Furthermore, the techniques adhere to the limitations imposed by the underlying PLC's architecture and standard.

In particular, PLC programs are subject to cyclic execution resulting in non-termination. Still, every execution through one cycle terminates and hence can be analyzed. Moreover, the standard forbids the implementation of recursive calls [Gro & Völ+ 22b]. In its current state, the PAF does not support all available language features of the [Int 14]. For instance, using arrays or pointers is not yet supported but could be extended. Statically allocated memory, however, can be modeled by flattening the arrays. While the PAF is able to analyze loops other than the naturally occurring execution cycle of the PLC program [Gro & Völ+ 22b], no additional heuristics such as loop invariant generation have been implemented, hence may rendering the analysis of loops in specific examples as intractable. Some of the benchmarks use the timer capabilities of the [Int 14], which are modeled non-deterministically using an over-approximating representation of timers from [Adi & Dar+ 14].

Last but not least, control tasks are usually distributed in the context of Industry 4.0, yet most often still coordinated centrally [Bre & Buc+ 19]. While multiple PLCs for each control task exist, they are coordinated centrally by one overarching controlling PLC. Despite that, the distributed control task is modeled as one compositional, classic PLC program, in which the other control tasks are incorporated as FBs and executed on a single PLC controller in the benchmarks [Gro & Völ+ 22b]. This neglects the influences of different times and latencies introduced due to the communication between each controlling PLC [Gro & Völ+ 22b]. It is assumed 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 [Gro & Völ+ 22b].

To this end, the contributions of this thesis include

- heuristics for the scalability of the existing TSG for PLC software,
- the reuse of symbolic summaries during TSG of reconfigured PLC software with the goal of reducing analysis time,
- and the concept of four-way forking for TSA of reconfigured PLC software

and are evaluated on selected domain-specific benchmarks of varying difficulty, such as the PLCopen Safety suite and the Pick and Place Unit (PPU).

1.4.3 Outline

Figure 1.16 gives a graphical overview of the implementation contribution of this thesis and their interplay and aligns them with the subsequent chapters. The input

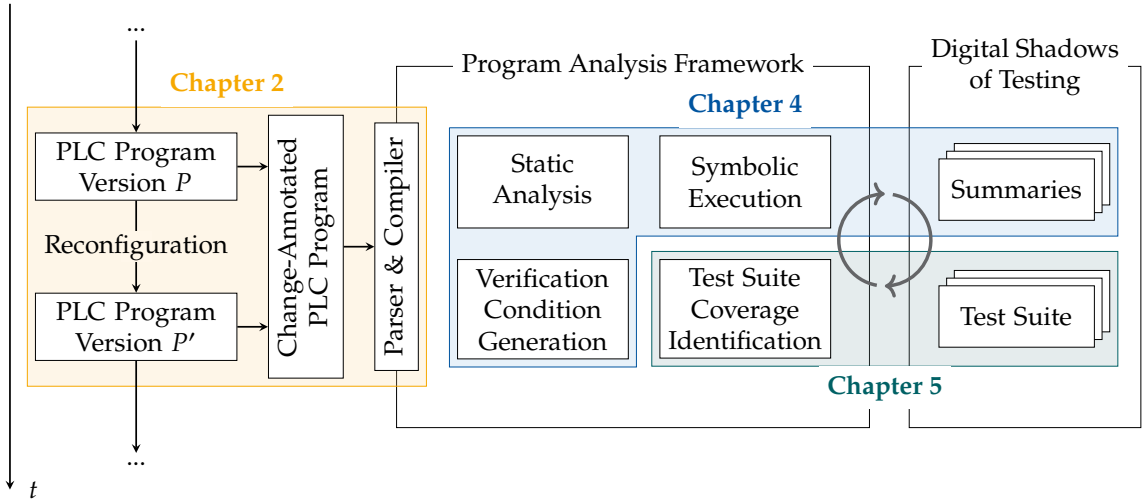


Figure 1.16: Overview of the implementation contribution of this thesis.

of the developed PAF is a change-annotated PLC program, which is parsed and compiled into a textual IR and represented as a CFG. Chapter 2 gives an introduction to PLCs and their peculiarities, presents the IR, gives a short digression to symbolic program analysis and their design space, discusses the unification of program versions in a CAP, and motivates how satisfiability modulo theories (SMT) formulas can be solved efficiently. The subsequent Chapter 3 gives insights into the field of TSG and TSA in regression analysis and discusses related work in

the field of PLC verification and testing. Chapter 4 deals with the background on TSG, the realization of summarization of FBs, and the applicability of summaries. It is represented by the blocks static analysis (SA), verification condition generation (VCG), symbolic execution (SE), and the summaries of Figure 1.16. Chapter 5 focuses on the TSA and extends the SE from Chapter 4. This chapter explains the underlying theory behind the test suite coverage identification depicted in Figure 1.16 and proposes shadow symbolic execution (SSE) as a solution to TSA. In Chapter 6, the two techniques proposed in this thesis are evaluated on benchmarks of varying sizes under different heuristics. Last, Chapter 7 concludes this thesis and gives suggestions for future work.

2 Preliminaries

This chapter gives an introduction to PLCs, their execution model, and an IR for ST. Following this, the design principles of SE and relevant concepts for the methods in Chapter 4 and Chapter 5 are presented.

2.1 Programmable Logic Controllers

PLCs are specially designed control hardware used in industrial automation [Tie & Joh 09] to control, operate, supervise, and monitor highly complex automation processes [Bia 16].

As PLCs are still ubiquitous in today's industrial control applications [Boh 21], they are subject to the IEC 61131 standard [Int 14]. The standard defines requirements toward both hardware and software aspects of the PLCs [Boh 21] for the prevention of personal injury caused by machines in the production process [Tie & Joh 09]. Figure 2.1 shows a schematic view of the execution model of the PLC and its interaction with its environment. PLCs adhere to a cyclic execution model.

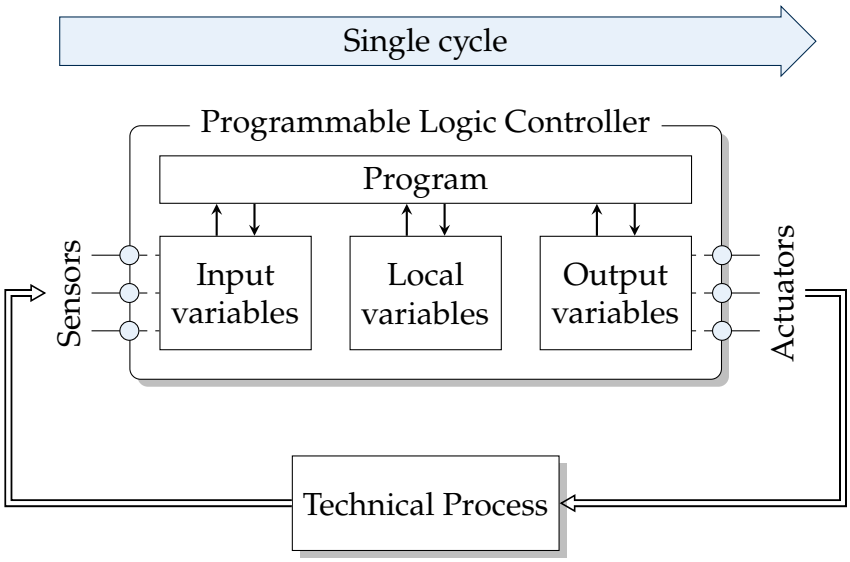


Figure 2.1: Schematic view of a PLC interacting with its environment.
Figure adapted from Figure 2.1 in [Boh 21].

At the beginning of an execution cycle, the inputs from the environment provided

by the sensors of the technical process are read and stored in the designated input variables of the PLC. For the duration of the execution cycle, the PLC will work on this copy of input variables and ignore any new values provided by the sensors until the beginning of the next cycle.

Next, the execution of the logic control program takes place. To update its execution state, the program may use local variables serving as internal memory, whose valuations persist across subsequent execution cycles. An execution cycle ends when the program terminates. Upon termination, the computed valuations of the output variables are used to update the outgoing connections to the actuators in order to invoke the desired behavior of the controlled plant. Between updates, the outputs retain their valuations throughout the subsequent cycles [Boh 21].

2.1.1 Program Organization Units

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 instruction that operates on this interface [Tie & Joh 09]. The IEC 61131 standard [Int 14] distinguishes between three types of POUs, namely functions, function blocks, and programs.

Function Functions are parameterizable POUs without local variables or FB instances. They are, therefore, "stateless" and always provide the same result for the same input parameters. Since there is no internal state, it is impossible to instantiate functions [Bia 16] as part of a POU's interface declaration, and an invocation requires passing all parameters directly to the function.

Function Block FBs are parameterizable POUs with local variables and FB instances. Instances of FBs are created as local variables of POUs. FBs retain the valuations of their internal and external variables between invocations and are, thus, "stateful". The passing of arguments is optional when invoking FBs [Boh 21].

Unlike functions, an instance of an FB is part of the parent POUs' local variables, and initialization only takes place for the first call of an instance. As internal and external variable valuations are retained between invocations, calling an FB without assigning new arguments to its input parameters will use the currently assigned values.

Typically, timer FBs are configured this way by passing arguments for all parameters and only partially assigning values to input parameters in subsequent calls [Boh 21].

Program This POU type represents the main entry point in each cycle and adheres to the schematic view of Figure 2.1. In addition to the local variables of FBs, programs have hardware inputs and outputs [Bia 16]. All variables of the overall program to which physical addresses, i.e., inputs and outputs of the PLC, are

assigned must be memory-mapped in this POU or the respective configuration [Tie & Joh 09].

2.1.2 Programming Languages

As PLCs are widely used in industrial automation, the IEC 61131 standard defines five distinct programming languages for implementing the functionality of a POU to account for the various programming paradigms [Bia 16]: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Function Block Diagram (FBD), and Sequential Function Chart (SFC) [Tie & Joh 09].

The choice of language depends on the tasks and areas of applications at hand and differs significantly [Tie & Joh 09]. Usually, lower-level and algorithmic functionality are implemented in textual languages, whereas the overall project structure is implemented in a graphical language [Boh 21].

As all of the IEC 61131 programming languages can be translated into each other [Boh 21], verification workflows make use of an IR [Dar & Maj+ 16]. The high-level textual language ST is introduced and used throughout this thesis for illustration purposes [Tie & Joh 09].

Structured Text

ST is a high-level textual language for imperative, procedural PLC programming [Boh 21] which resembles PASCAL [Tie & Joh 09]. A compositional example program consisting of two POUs, a program P and an FB Fb, is shown in Figure 2.2.

Example 2.1: Explanation of Figure 2.2

The main program, P, instantiates an FB of type Fb and invokes it in the body. The input variable is passed as an argument via assigning $x := a$, and the output is written via $z => b$. The callee f computes its output z depending on the valuation of the input of x and the internal state of y. It uses y as an internal counter variable and returns $z := TRUE$ if the valuation reaches the value 3. The value of the local variable y of the callee f is initialized with 0. User-defined initialization is achieved by using the allocation operator such as $y : INT := 1;$.

Default Initialization of Variables

At the start of the program, all variables are assigned initial values. These valuations depend on the specification by the developer made in the corresponding declaration parts of the POUs [Tie & Joh 09]. The IEC 61131 ensures that all elementary data types have predefined initial values [Int 14]. Elementary data types, for instance, are initialized with 0 for numeric values and *false* for Boolean values. This is also

```
PROGRAM P
VAR_INPUT
    a : INT;
END_VAR
VAR
    f : Fb;
END_VAR
VAR_OUTPUT
    b : BOOL;
END_VAR
    f(x:=a,z=>b);
END_PROGRAM
```

```
FUNCTION_BLOCK Fb
VAR_INPUT
    x : INT;
END_VAR
VAR
    y : INT;
END_VAR
VAR_OUTPUT
    z : BOOL;
END_VAR;
    IF x >= 32 THEN
        y := y + 1;
    ELSE
        y := y;
    END_IF;
    IF y >= 3 THEN
        z := TRUE;
        y := 0;
    ELSE
        z := FALSE;
    END_IF;
END_FUNCTION_BLOCK
```

Figure 2.2: A program POU and an FB POU.

termed 0-default initialization throughout this thesis. For example in Figure 2.2, the value of the local variable y of the callee f is initialized with 0. User-defined initialization is achieved by using the allocation operator such as $y : INT := 1;$.

2.2 Intermediate Representation

Each POU is compiled into a type-representative module representing the program semantics in a goto-based textual IR. The supported expressions by these semantics are due to [Nie & Nie 92; Nie & Nie 20] and presented in Definition 2.1.

Definition 2.1: Expression [Nie & Nie 92; Nie & Nie 20]

An expression $e \in \boldsymbol{E}$ is either an arithmetic expression $a \in \boldsymbol{E}_A \subseteq \boldsymbol{E}$ or a Boolean expression $b \in \boldsymbol{E}_B \subseteq \boldsymbol{E}$.

$$
\begin{aligned}
e &::= a \mid b \\
a &::= i \mid v_a \mid -a \mid a_1 \, op_a \, a_2 \\
b &::= t \mid v_b \mid \neg b \mid a_1 \, op_r \, a_2 \mid b_1 \, op_b \, b_2
\end{aligned}
$$

The meta-variable $i \in \mathbb{I}$ represents integers and ranges over the implementation-specific integer data type $\mathbb{I}$ of the programming language used for Z3 [dMou & Bjø 08], as it is used as implementation backend for expressions within this thesis. The meta-variable $t \in \mathbb{B}$ represents the Boolean truth values *true* and *false*. The arithmetic and Boolean variables are denoted through the meta-variable $v_a \in \boldsymbol{V}_A \subseteq \boldsymbol{V}$ or $v_b \in \boldsymbol{V}_B \subseteq \boldsymbol{V}$, respectively. The supported subset of operators with their corresponding semantics from ST [Int 14] are $op_a \in \{**, *, /, \text{MOD}, +, -\}$ for arithmetic expressions and $op_r \in \{<, >, \leq, \geq, =, <>\}$ and $op_b \in \{\text{AND}, \text{XOR}, \text{OR}\}$ for Boolean expressions [Tie & Joh 09]. Parentheses, "(" and ")", are resolved during compilation from ST to the IR via the precedence of operators.

The ST statements are compiled into instructions of the IR using the expressions defined in Definition 2.1.

Definition 2.2: Instruction

An instruction has one of the following forms:

$$
\begin{aligned}
I ::= \; & \textbf{goto}\, b_\ell \mid \textbf{sequence}(I_1, I_2) \mid \textbf{assign}(v, e) \\
& \mid \textbf{ite}(b, \textbf{goto}\, b_{\ell_1}, \textbf{goto}\, b_{\ell_2}) \mid v_1, \ldots, v_n := \textbf{call}\, G(e_1, \ldots, e_m)\,.
\end{aligned}
$$

Throughout this thesis, the call instruction $v_1, \ldots, v_n := \textbf{call}\, G'(e_1, \ldots, e_m)$ is lowered to a sequence of pre- and post-assignments without the loss of general-

ity [Cla & Hen+ 18]. During compilation calls are completed, i.e., additional pre- and post-assignments are introduced such that all variables are always initialized. It is determined which inputs and outputs are read and written and the respective assignments are augmented, enforcing the same order as in the underlying interface of the compiled, type-representative module. The cyclic execution semantic of a PLC program is not explicitly captured via an instruction. Instead, it is implicitly captured by Definition 2.3. Possible execution paths through this IR are of interest to analyze a program. Therefore, the IR representing the program structure can be formalized as a flow graph. Here, the instructions are labeled and assigned to vertices. In principle, assigning the instructions to the edges is equivalent, and the appropriate choice of formalization depends on the particular analysis [Ste 93]. The edges between the vertices represent the intra- and interprocedural flow of control.

Definition 2.3: Control-Flow Graph [All 70]

A *control-flow graph* (CFG) is a tuple $G = (\boldsymbol{V}, \boldsymbol{V}_{input}, (B, E), b_{\ell_e}, b_{\ell_x})$, where

- $\boldsymbol{V}$ is an ordered finite set of variables,
- $\boldsymbol{V}_{input} \subseteq \boldsymbol{V}$ is an ordered finite set of input variables,
- (B, E) is a directed graph with
  - vertices B representing labeled blocks $b_\ell \in B$,
  - edges $E \subseteq B \times B$ modeling the potential transfer of control from the end of the block to the beginning of the next block,
- $b_{\ell_e} \in B$ is a unique entry block,
- $b_{\ell_x} \in B$ is a unique exit block, and without loss of generality $b_{\ell_e} \neq b_{\ell_x}$.

The finite set of variables $\boldsymbol{V} = \boldsymbol{V}_{input} \uplus \boldsymbol{V}_{local} \uplus \boldsymbol{V}_{output}$ results from the disjoint set union of the input, local, and output variables of all referenced type-representative CFGs that are connected to this CFG. Each labeled block $b_\ell \in B$ can contain zero or more instructions. A single block encoding of the control flow is enforced for the SE in the subsequent Chapters 4 to 5, therefore disallowing sequence of instructions as commonly referred to as basic block encoding without the loss of generality. This implies that an instruction within a vertex b_ℓ of G is uniquely identifiable by its label ℓ.

Definition 2.4: Program

A *program* is a pair $\mathcal{P} = (G, \mathcal{G})$, where

- $G \in \mathcal{G}$ is the CFG representing the program POU,

- $\mathcal{G}$ is a set of CFGs representing POUs referenced by the PLC's program POU.

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 FBs occurring in the program.

Example 2.2: Control-Flow Graph

Figure 2.3 shows the compiled running example. $\mathcal{P} = (P, \{P, Fb\})$, where $P = (\{a, b, f.x, f.y, f.z\}, \{a\}, (B, E), b_0, b_{13})$. Do note that variables such as $f.x$ do not represent qualified names anymore but are instead flattened as the internal memory of FBs are lowered during the compilation to regular procedures which operate on references to the blocks' variables [Boh 21]. Hence the qualified names [Aho & Set+ 86] such as $f.x$, where f and x are identifiers, are lowered to a single identifier $f.x$ and therefore represent a single flattened variable.

Representation of Callees

Several ways of modeling callees exist, enabling different precisions for the analyses [Nie & Nie+ 99]. A callee is a function that is called by another function, the caller. The algorithms of this thesis were designed with function cloning in mind. Cloning provides some form of context sensitivity through the use of scopes but avoids redundancy as introduced by function inlining.

A benefit of cloning is the avoidance of interprocedural invalid paths. Nevertheless, high nesting depths may lead to an exponential blow-up. A downside of function cloning is that it does not work on (mutually) recursive functions. However, the PLC standard [Int 14] restricts the programmer from implementing recursion. While avoiding interprocedural invalid paths under-approximates the program's behavior, the analysis is still compositional in a broader sense as, for instance, the summarization from Chapter 4 abstracts from the actual context.

2.3 Symbolic Program Analysis

Next, some common concepts from the symbolic program analysis's perspective are explained to resolve ambiguity [Aho & Set+ 86]. Figure 2.4 illustrates the relationship between these concepts. Variables refer to particular locations within the memory, called the *store*. A state is a mapping from locations in the store to values, i.e., a state maps "l-values" to their corresponding "r-values". An l-value always has a defined location within the memory and thus can be referenced. However, an r-value is an expression that is not an l-value. In the context of symbolic

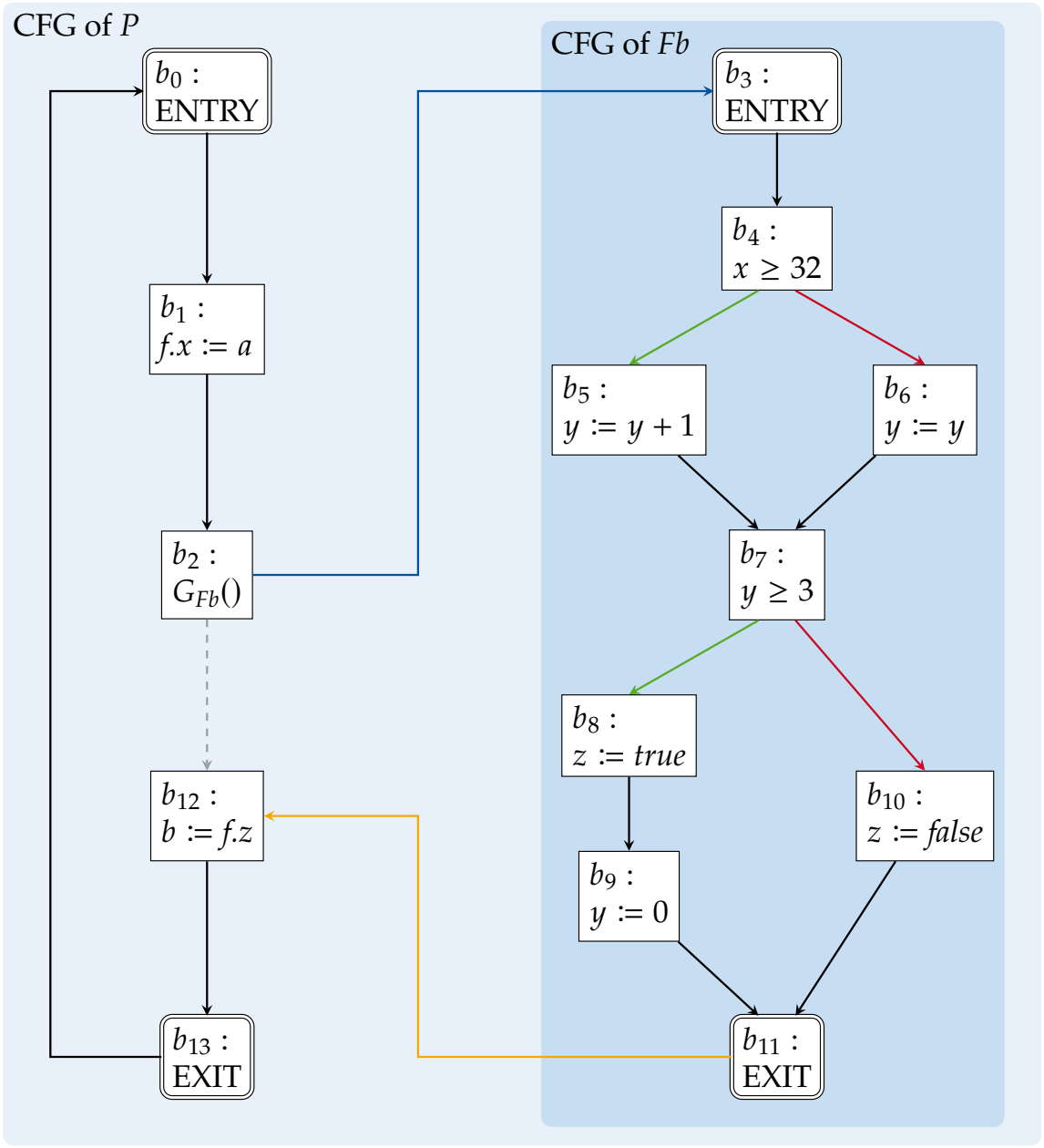


Figure 2.3: Graphical representation of the compiled running example.

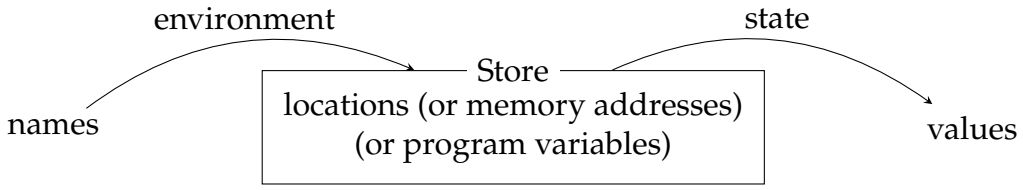


Figure 2.4: Relationship between the environment, store, and state [Aho & Set+ 86].
Figure adapted from Figure 1.8 in [Aho & Set+ 86].

program analysis [Bal & Cop+ 18], the state refers to an execution state, whereas the store describes this mapping. Usually, the semantics use two mappings, one for the environment $\epsilon : \boldsymbol{V} \to \text{Loc}$, which maps names describing variables to locations $\text{Loc} \coloneqq \mathbb{N}$, and one for the state $\iota : \text{Loc} \to \boldsymbol{D}$, which maps locations to values of the respective domain $d \in \boldsymbol{D}$, i.e., integers $i \in \mathbb{I}$ and Boolean truth values $t \in \mathbb{B}$ in this thesis, as depicted in Figure 2.4. This thesis abstracts from this concept of composition and directly maps names to their values which is common in symbolic program analysis [Nie & Nie+ 99]. This facilitates the notation as compile-time names are equivalent to run-time locations, i.e., variables [Aho & Set+ 86], and also coincides with how POUs are compiled in the IR of this thesis (see Example 2.2). The definitions of the concrete and symbolic store used in symbolic program analysis with regard to this thesis are given in Definition 2.5.

Definition 2.5: Concrete and Symbolic Store

A concrete store $\rho : \boldsymbol{V} \to \boldsymbol{D}$ is a mapping from variables $v \in \boldsymbol{V}$ to concrete values $d \in \boldsymbol{D}$. A symbolic store $\sigma : \boldsymbol{V} \to \boldsymbol{\Sigma}$ is a mapping from variables $v \in \boldsymbol{V}$ to symbolic expressions $\gamma \in \boldsymbol{\Sigma}$.

A symbolic expression $\gamma \in \boldsymbol{\Sigma}$ is essentially just an expression $e \in \boldsymbol{E}$ defined over additional names not defined in the IR of the analyzed PLC program. Variables can be read and written to the respective stores, and manipulation is denoted by either accessing, i.e., reading from the store, or substitution, i.e., writing to the store.

Definition 2.6: Substitution

Substitution of a variable $u, v, x, y \in \boldsymbol{V}$ with a concrete value $d \in \boldsymbol{D}$ or a symbolic expression $\gamma \in \boldsymbol{\Sigma}$ is defined over the concrete and symbolic store as follows

$$\rho[d/v](u) \coloneqq \begin{cases} d & \text{if } u = v, \\ \rho(u) & \text{otherwise} \end{cases} \qquad \sigma[\gamma/y](x) \coloneqq \begin{cases} \gamma & \text{if } x = y, \\ \sigma(x) & \text{otherwise}, \end{cases}$$

where all occurrences of v and y are substituted in the concrete and symbolic store by d and γ, respectively.

Throughout this thesis, the substitution operation of Definition 2.6 is generalized to the short-hand update functions $\rho[v \mapsto d]$ and $\sigma[y \mapsto \gamma]$.

Furthermore, the notations $\text{eval}_\rho(e)$ and $\text{eval}_\sigma(e)$ are used to denote the evaluation of the expression $e \in \boldsymbol{E}$ with regards to the evaluation relation $\langle e, \rho \rangle \rightarrow d$ for the concrete store and $\langle e, \sigma \rangle \rightarrow \gamma$ for the symbolic store defined in Appendix A, respectively. The definitions for the concrete and symbolic evaluation functions are given in the Definitions 2.7 to 2.8.

Definition 2.7: Concrete Evaluation Function

Given a concrete store ρ, the evaluation function $\text{eval}_\rho \colon \boldsymbol{E} \rightarrow \boldsymbol{D}$ is defined recursively and assigns to each expression $e \in \boldsymbol{E}$ a value from the respective domain $d \in \boldsymbol{D}$, integer or Boolean.

Definition 2.8: Symbolic Evaluation Function

Given a symbolic store σ, the evaluation function $\text{eval}_\sigma \colon \boldsymbol{E} \rightarrow \boldsymbol{\Sigma}$ is defined recursively and assigns to each expression $e \in \boldsymbol{E}$ a symbolic value from the respective domain $\gamma \in \boldsymbol{\Sigma}$, integer or Boolean.

Static and Dynamic Symbolic Execution

A branch of symbolic program analysis deals with static and dynamic symbolic execution (SE) [Bal & Cop+ 18]. Before looking at the differences between static and dynamic SE, a high-level description of how SE works is given.

SE automatically explores program paths by executing the underlying CFG of the IR for the respective PLC program using symbolic values for the input variables. In case concrete valuations are also considered, SE is called concolic execution (CE), a portmanteau of concrete and symbolic execution.

Definition 2.9: Path

A path through a CFG is a sequence of $m > 0$ edges $e_1, \ldots, e_m$ such that given basic blocks $b_i, b_j, b_k, b_l \in B$ and $0 < n < m$ then $e_n := (b_i, b_j)$ and $e_{n+1} := (b_k, b_l)$, then $b_j \equiv b_k$.

A path is feasible if there exists an input to the program that "covers" the path, i.e., when the program with that input is executed, the corresponding path is taken. A path is infeasible if there exists no input that covers the path.

The execution strategy of SE can adhere to two different strategies [Kuz & Kin+ 12]. Either a path is completely executed from the entry to the exit of the CFG

in a depth-first exploration ("DART-style" [God & Kla+ 05]), or the execution is more shallow and performed in a breadth-first exploration ("EXE-style" [Cad & Gan+ 08]). This thesis uses the "EXE-style" execution over the "DART-style" execution as it gives more fine-grained control over the execution. SE begins from an *initial* execution state and unfolds the set of reachable execution states in a certain number of "steps". This unfolding leads to a partition of the input space resulting in an enumeration of all feasible paths. In general, a CFG over-approximates the executable behavior.

Definition 2.10: Execution State [Bal & Cop+ 18]

An execution state $s = (b_\ell, \rho, \sigma, \pi)$, where

- $b_\ell \in B$ is next vertex of the CFG to execute,
- ρ is a *concrete store* that maps variables $v \in \boldsymbol{V}$ to expressions over concrete values,
- σ is a *symbolic store* that maps variables $v \in \boldsymbol{V}$ to expressions over concrete and symbolic values,
- π denotes the *path constraint* representing a set of conditional expressions taken on the execution path up to vertex b_ℓ.

The path constraint π characterizes a set of input parameters for which the program executes along the path [God 07]. The branching conditions are recorded in the path constraint π whenever the execution "forks" at a branch. In the domain of PLC software, the path constraint π also carries constraints on local variables resulting from decisions at branches. A constraint solver is used to decide the feasibility of the recorded paths and can generate a satisfying assignment that can be used as a witness or test case for the particular path.

While in static SE, the whole program source code is represented as a formula by computing the strongest post-conditions beginning from the entry of the CFG, dynamic SE actually executes the program using concrete valuations while maintaining symbolic valuations for input-dependent choices of paths at branches. Consequently, an advantage of dynamic SE is that no false positives are derived, and one can always obtain useful partial results [Bal & Cop+ 18].

Still, in order to guarantee the absence of bugs, all paths must be enumerated. As execution goes on, the number of enumerated paths can become too large to handle efficiently, resulting in the path explosion problem.

2.4 Design Principles of Symbolic Execution

Path explosion is a problem occurring during SE [Bal & Cop+ 18]. There are four major ways to tackle the problem of path explosion in SE [Bey & Lem 16]:

1. Search heuristics for achieving a high level of branch or path coverage as fast as possible
2. Compositional symbolic execution, creating summaries of functions or paths and reusing them instead of recomputing already explored states
3. Handling of unbounded loops
4. Using interpolants for tracking reasons why a particular path is infeasible

These solutions are tightly coupled with how the underlying SE engine and algorithms are designed [Kuz & Kin+ 12; Kuz & Kin+ 14].

2.4.1 Handling of Loops and Recursion

Standard techniques which allow the analysis of loops and recursion are unrolling and function call inlining [Kuz & Kin+ 14; Bec & Ulb+ 15]. A peculiarity of PLC programs is that recursive call chains are forbidden by design as defined in the standard [Int 14]. Static unrolling of loops in PLC programs is usually done up to a specific bound by statically rewriting the underlying IR [Bec & Ulb+ 15].

In this thesis, no techniques or heuristics have been investigated that deal with the problem of handling loops and recursions efficiently. In fact, the assumption is made that the programs under analysis do not contain any unbounded or "problematic" loops. As PLC programs are employed to solve time-critical tasks, and the standard requires the programs to finish within a specified cycle time [Int 14], it is, therefore, reasonable to assume that programs usually do not contain loops with an unbounded number of iterations [Bec & Ulb+ 15].

The algorithms of Chapter 4 and Chapter 5 explore loops as long as they cannot prove the infeasibility of the condition.

2.4.2 Avoiding the Encoding of Infeasible Execution Paths

In general, symbolic program analysis avoids encoding infeasible execution paths by checking for satisfiability at points in the program where the control flow branches. By not considering execution states that represent infeasible paths, the problem of path explosion is combatted by investing solving time earlier in the execution [Kuz & Kin+ 14]. In this thesis, feasibility checking is also performed when "forking" and is explained in detail in Chapter 4.

2.4.3 Merging of Execution Paths

State merging combines paths into a state [Bal & Cop+ 18] and is a suitable technique to combat the path explosion problem [Kuz & Kin+ 14]. Unlike in SA, where it is acceptable that the resulting merged state over-approximates the individual states that were merged, SE for TSG requires more precise information [Kuz & Kin+ 14].

A possible way to merge all execution paths without any over-approximation reaching a particular vertex is using the path constraint π to determine from which path the information reached this specific vertex.

Example 2.3: State Merging

Merging execution states in each cycle of a PLC program yields a linear path growth and is a desired technique to combat the path explosion problem. The tradeoff, however, is that state merging generates more complex execution states. Consider merging the following two execution states described by a path constraint and their concrete and symbolic valuations:

$$\texttt{merge}\left(\left\{\begin{matrix} x_0 \geq 0 \\ x \mapsto x_0 \\ y \mapsto 1 \end{matrix}\right\}, \left\{\begin{matrix} x_0 + 1 < 8 \\ x \mapsto 0 \\ y \mapsto 8 \end{matrix}\right\}\right) = \left\{\begin{matrix} x_0 \geq 0 \vee x_0 + 1 < 8 \\ x \mapsto \texttt{ite}(x_0 \geq 0, x_0, 0) \\ y \mapsto 1 \end{matrix}\right\}$$

Symbolic valuations are merged with the help of the `ite`-expression of the underlying SMT solver. Due to the efficient representation of expressions in the abstract syntax tree (AST) in Z3, the merged expressions are kept as compact as possible (without duplication) in memory, even in the presence of nested `ite`-expressions. Either valuation can be chosen when merging concrete valuations which leads to an under-approximation of the concrete store.

State merging can occur at different points during program execution. Either all paths are executed until the end of the cycle, and the subsequent cycle is explored with the merged execution state of all paths, or at each point where the control flow joins, the respective paths are merged. This is explained with the help of Example 2.4.

Example 2.4: Join Points during State Merging

This example shows how two merge-based strategies can be used during the exploration of the CFG.

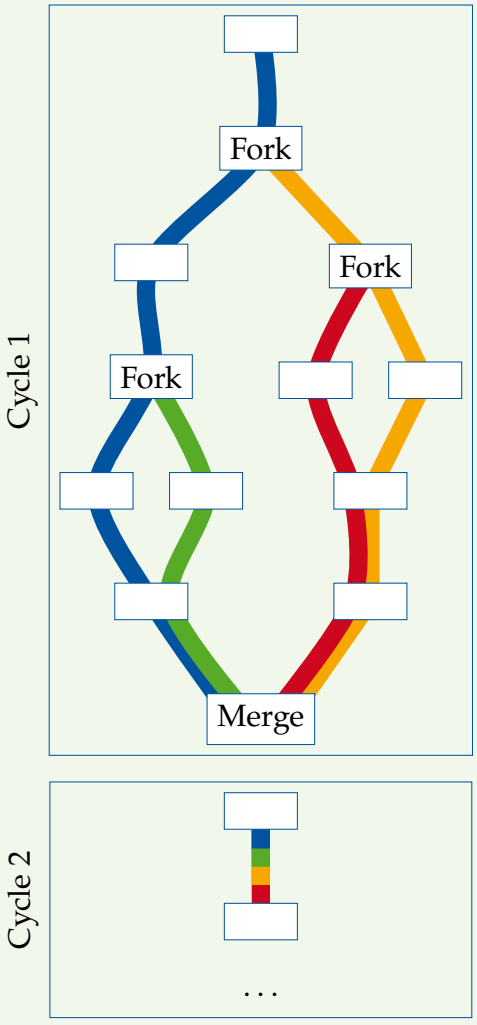


Figure 2.5: Merging at the end of the cycle.

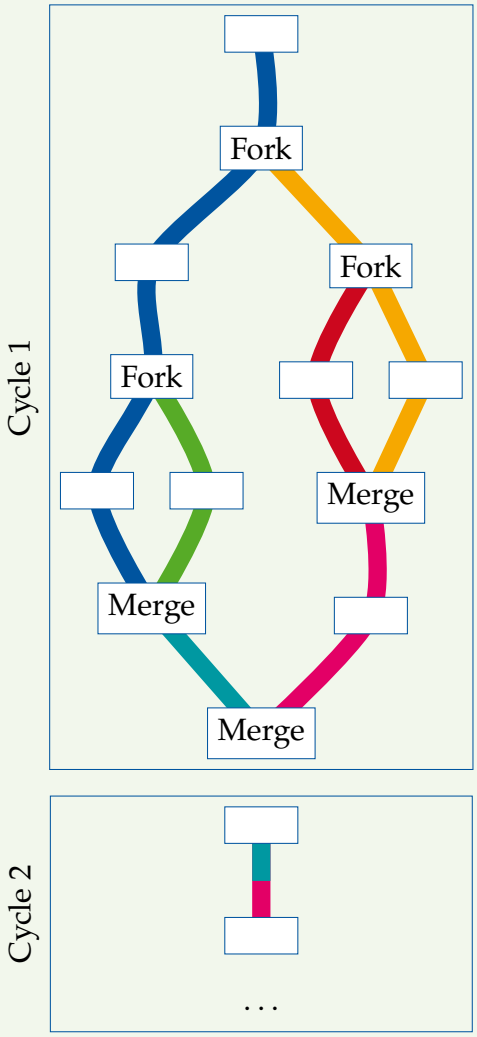


Figure 2.6: Merging at all join points.

Either way, merging reduces the number of execution states and combats the path explosion problem as an exponential growth with each cycle is prevented. However, merging execution states introduces more complex formulas and offloads the burden to the underlying SMT solver.

Merging states this way for every possible join point, i.e., where control flow merges in the graph, SE would become similar to VCG or bounded model checking (BMC), where the entire problem instance is encoded in one monolithic formula that is passed in full to a solver [Kuz & Kin+ 14]. However, encoding the problem instance as one monolithic formula also has downsides. The complete merging

sacrifices the advantages of continuously progressing, prioritizing execution states, and reaching coverage goals quickly, and thus more dynamic approaches are favorable [Kuz & Kin+ 14].

2.4.4 Dealing with Compositionality

While function call inlining is a simple and precise approach for interprocedural symbolic program analysis, it is also costly [Kuz & Kin+ 14]. Function cloning poses an alternative to function inlining and is used for the representation of callees throughout this thesis, as presented in Section 2.2.

Function cloning provides context sensitivity and therefore avoids the encoding of invalid interprocedural paths. However, high nesting depths can lead to an exponential blow-up, just as with function inlining. The drawback of not working on mutually recursive functions is neglectable for the programs in the PLC domain. This gives rise to the use of function summaries such that re-analyzing functions and FBs at every invocation can be avoided [God 07].

May and Must Summaries

The number of paths to be explored can be reduced by using *memoization* in the form of summaries [Can & God 19]. A summary is either of the form "may" or "must". While SA generates and memoizes over-approximations ("may" summaries), TSG computes under-approximations ("must" summaries) rooted in feasible executions.

Definition 2.11: Must Summary [God 07; God & Lah+ 11]

A must summary of a code fragment is of the form $\langle lp, P, lq, Q \rangle$, and implies that for every execution context satisfying the pre-condition P at lp in the program, there exists an execution that visits lq and satisfies the post-condition Q at lq.

A pre-condition defines an equivalence class of concrete executions [God 07]. A post-condition describes the valuations of the variables at the exit of the summary. Given a path constraint π_w and a compositional program P. The summary $\phi_{fb} = \bigvee_w \phi_w$ is computed by symbolically executing all feasible paths of the FB and generating a pre- and post-condition with regard to the input-output relation of the respective FB. Each disjunct ϕ_w represents one summarized path through the FB and is of the form $\phi_{w_{fb}} = pre_{w_{fb}} \wedge post_{w_{fb}}$, where $pre_{w_{fb}}$ is a conjunction of constraints on the inputs of FB while $post_{w_{fb}}$ is a conjunction of constraints on the outputs of FB [God 07].

Example 2.5: Must Summary

Consider the CFG of the FB depicted in Figure 2.3. While y is a local variable of the FB, it must be considered in the respective must summary. This clashes with the definition of preconditions consisting of only whole-program inputs [God 07] but is a necessary adaption to represent execution paths of stateful FBs correctly. A summary for the right-most outer path through the CFG of the FB is characterized by the following formula:

$$\phi_{w_{fb}} := \underbrace{\neg(x \geq 32) \wedge \neg(y \geq 3)}_{pre_{w_{fb}}} \wedge \underbrace{(z = \mathit{false})}_{post_{w_{fb}}} .$$

The mandatory versioning of variables, as presented in Section 4.1.5, is neglected for reasons of the example.

2.5 Unifying Program Versions via Change Annotations

Syntactic reconfigurations to industrial control software can be categorized following the concept in [Kuc & Pal+ 18], where a `change(old, new)` macro was introduced.

Definition 2.12: Change-Annotation Expression [Pal & Kuc+ 16; Kuc 16; Kuc & Pal+ 18]

A *change-annotation macro* `change(old, new)` in the source code of a PLC program is compiled into a binary *change-annotation expression* $\texttt{change}(e_1, e_2)$ which consists of two compiled expressions e_1 and e_2 of the same type corresponding to the expression in the old and new program, respectively. It naturally extends the set of expressions of the IR defined in Definition 2.1 as follows:

$$\begin{aligned} e &::= a \mid b \mid c \\ a &::= i \mid v_a \mid -a \mid a_1 \; op_a \; a_2 \\ b &::= t \mid v_b \mid \neg b \mid a_1 \; op_r \; a_2 \mid b_1 \; op_b \; b_2 \\ c &::= \texttt{change}(e_1, e_2) \; , \end{aligned}$$

where $c \in \boldsymbol{E}_C \subseteq \boldsymbol{E}$ is a change expression.

The following enumeration summarizes the available reconfigurations using the categorization of [Kuc & Pal+ 18] expressed in the ST syntax.

Addition of new functionality

1. Addition of an assignment:

```
x := change(old: x, new: e);
```

2. Addition of code in the new program version:

```
IF change(old: false, new: true) THEN
    code to be added
END_IF;
```

Modification of already existing functionality

1. Modifying the right-hand side of an assignment:

```
x := change(old: e1, new: e2);
```

2. Modifying the arguments of a function block invocation:

```
f(x := change(old: e1, new: e2),...);
```

3. Modifying a conditional expression:

```
IF change(old: e1, new: e2) THEN
```

Deletion of functionality

1. Deletion of an assignment:

```
x := change(old: e, new: x);
```

2. Deletion of code in the new program version:

```
IF change(old: true, new: false) THEN
    code to be deleted
END_IF;
```

A significant benefit of the change-annotation macro is that it keeps the correspondence between the old and the new program versions intact. It was therefore chosen for analyzing the semantic effects of the implication introduced by syntactical reconfigurations [Gro & Völ+ 22b].

Example 2.6: Reconfiguration applied to the Running Example

Consider the following syntactic change applied to the FB of Figure 2.2.

```
1  FUNCTION_BLOCK Fb
2  // Interface omitted.
3  IF x >= 32 THEN
4      y := y + 1;
5  ELSE
6      y := y;
7  END_IF;
8  IF y >= 3 THEN
9      z := TRUE;
10     y := 0;
11 ELSE
12     z := FALSE;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 2.1: Callee POU before reconfiguration.

```
1  FUNCTION_BLOCK Fb'
2  // Interface omitted.
3  IF x >= 32 THEN
4      y := y + change(1, 2);
5  ELSE
6      y := y;
7  END_IF;
8  IF y >= 3 THEN
9      z := TRUE;
10     y := 0;
11 ELSE
12     z := FALSE;
13 END_IF;
14 END_FUNCTION_BLOCK
```

Listing 2.2: Callee POU after reconfiguration.

Reconfigurations are translated into syntactic reconfigurations applied to the source code of the PLC program with the help of the `change(old, new)` macro.

2.6 Formal Reasoning with the SMT Solver Z3

The algorithms presented throughout this thesis heavily rely on using the SMT solver Z3 [dMou & Bjø 08]. Satisfiability is the problem of determining whether a formula ϕ has a satisfying assignment, i.e., a model. SAT solvers check the satisfiability of propositional formulas. A propositional formula contains either *true* or *false* atomic propositions; accordingly, a model is a truth assignment to these Boolean variables.

An SMT solver checks the satisfiability of first-order formulas in some first-order decidable theories, such as linear arithmetic. The goal is to find a model that assigns values to variables and interpretations to all occurring predicates in the formula ϕ within the respective theory T. Z3 is such an SMT solver, and its API is used to model this thesis's internal IR. Figure 2.7 gives a high-level overview of how an SMT solver is used during SE.

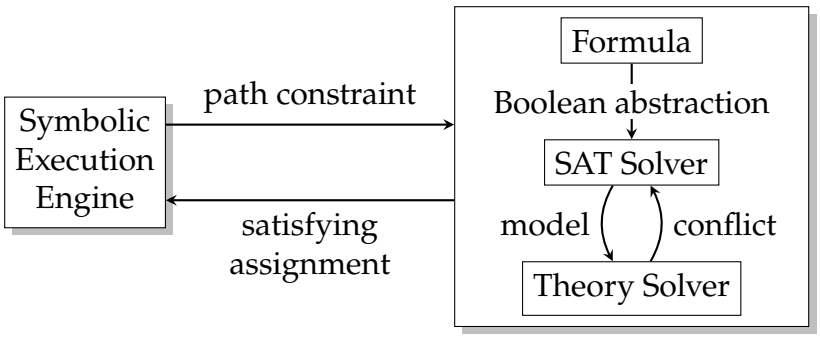


Figure 2.7: Interplay of SE and SMT solving with Z3.

Incremental Interface The Z3 API [1] provides two interfaces for adding and removing constraints during incremental solving: push/pop and assumptions. Do note that being able to add constraints does not make the solver incremental. The incrementality stems from storing and restoring the solver's state by a call to push and pop, respectively. This way, an initial set of assertions can be checked for satisfiability, and other additional assertions and checks may follow.

In case the addition of an assertion leads to unsatisfiability, the lemmas that were learned and are not valid can be removed with a pop, and a previous state is restored. This can be tedious when checking many potential execution paths for satisfiability. A more flexible approach poses the assumptions interface. However, for each assertion, a fresh Boolean assumption literal must be introduced, which implies the assertion.

During solving, the individual assertions can be considered for checking by toggling the corresponding Boolean assumption literals. A check with assumptions also has the benefit that in case of unsatisfiability, an unsatisfiable core can be obtained. The unsatisfiable core contains a subset of the assertions that the SMT solver used to deduce unsatisfiability.

[1] https://z3prover.github.io/api/html/index.html

Literature Review

3

"If I have seen further it is by standing on ye sholders of Giants."
— Sir Isaac Newton, *Letter to Robert Hooke (15 February 1676)*

This chapter overviews the state of the art of the main techniques behind the algorithms contributed by this thesis in Chapter 4 and Chapter 5 and discusses their relevance in program analysis. Next, a digression into regression analysis is

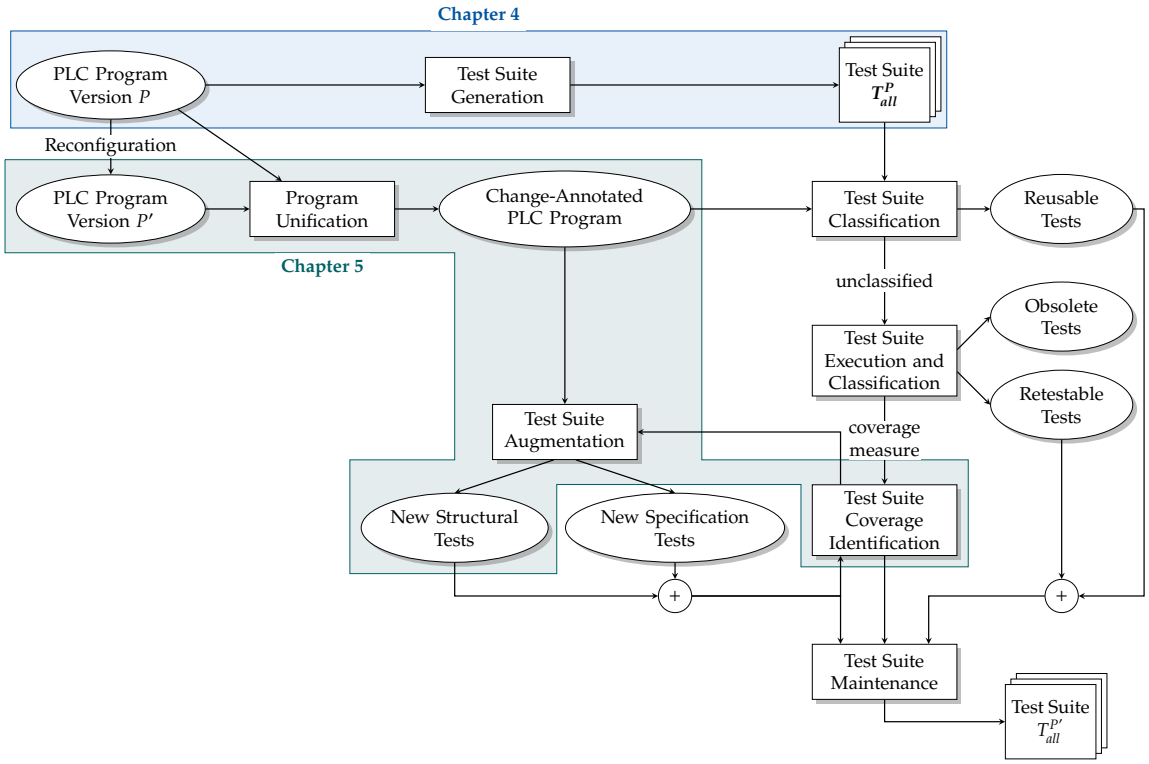


Figure 3.1: Overview of the regression testing pipeline.
Figure adapted from Figure 3 in [Leu & Whi 89].

presented, and relevant alternative work from verification and testing relying on this technique is discussed. Finally, the related work of verifying and testing the PLC software is presented.

3.1 Test Suite Generation via Symbolic Execution

SE is a precise and systematic program analysis technique that can be used for the automated generation of test cases [Kin 76; Dar & Kin 78] and has been used ever since for a variety of symbolic program analyses [Cad & Sen 13]. While the core technique remains the same among a variety of contributions to the state of the art [Sen & Mar+ 05; Cad & Gan+ 08; Cad & Dun+ 08; Bur & Sen 08; God & Lev+ 12], the following sections solely focus on the design principles [Kuz & Kin+ 12] mentioned in Section 2.4 relevant for the algorithms presented in Chapter 4 and Chapter 5 of this thesis. A recent overview of applications, tools, and optimizations of SE can be found in [Bal & Cop+ 18].

3.1.1 Compositionality and State Merging

Compositionality and state merging are vital techniques to combat path explosion [Bal & Cop+ 18].

Compositional symbolic execution (CSE) for TSG was first introduced in SMART [God 07], an extension of the directed automated random testing tool DART [God & Kla+ 05]. SMART computes function summaries during SE by testing functions in isolation when they are encountered in a top-down search on a demand-driven basis [God 07]. Function summaries are expressed over input pre-conditions and output post-conditions and computed by traversing each path within the respective function [God 07]. The input pre-conditions are obtained by simplifying the conjunction of branch conditions concerning the function inputs on that path. The output post-conditions are obtained by considering the conjunction of constraints written during the execution of the function [God 07]. Whenever a function is encountered during the search of the SE, SMART checks if an applicable summary already exists. If it does, the execution context is updated with the summarized contents, and the search skips the function's execution. Otherwise, the function is explored under the current calling context while SMART tries to generate a function summary [God 07].

The limitations of SMART were addressed in a subsequent publication by implementing the possibility of incrementally constructing partial function summaries to avoid the analysis of unnecessary and hard-to-analyze functions whose constraints were outside of the SMT solvers' theory [Ana & God+ 08].

Summarization was also addressed in [Lin & Mil+ 15], where it was generalized to any code fragments not explicitly limited to functions. Unlike in SMART, summarization takes place before SE. The paper evaluated three strategies of differing summarization granularity alongside the standard CSE. It was shown that an acyclic summarization strategy, "LASUM", was the most effective in reducing the average computation time of CSE [Lin & Mil+ 15]. This heuristic summarizes the contents of the most inner loop by breaking the summarization of the outer loop down to the summarization before, during, and after the inner loop. This mitigates the path explosion problem for nested loops compared to the other summarization

strategies evaluated in the paper by pruning redundant calls to the SMT solver [Lin & Mil+ 15].

MultiSE proposes a new technique for merging execution states incrementally using value summaries [Sen & Nec+ 15]. In contrast to conventional merging in SE, value summaries avoid redundant executions by sharing values along multiple execution paths [Sen & Nec+ 15]. By avoiding auxiliary symbolic values, while merging, MultiSE can handle function calls more efficiently by using fewer calls to an SMT solver. The evaluation has shown that this leads to a significant speedup between conventional merging in SE and MultiSE.

The algorithms of this thesis use the if-then-else expression `ite` of the underlying SMT solver for merging symbolic expressions at feasible join points [Boh & Sim+ 16]. While value summaries and `ite`-expressions are similar, the former does not require simplification as no nested `ite`-expressions occur during the application of MultiSE [Sen & Nec+ 15]. In general, one is interested in reducing the solving cost induced by merging while at the same time leveraging the advantages of SE [Avg & Reb+ 14; Kuz & Kin+ 14].

3.1.2 Incremental Solving, Search Heuristics, and Memoization

The work of [Lin & Mil+ 15] is succeeded by [Lin & Mil+ 16; Lin 17] in which the restriction of summarization from functions to arbitrary summarization of code fragments was alleviated by introducing incremental solving. A weakness of CSE is the loss of context during summarization, resulting in a bottleneck in the exploration of complex systems [Lin & Mil+ 16]. The paper proposes a combination of incremental solving and CSE to mitigate this weakness by compensating with the incremental assumption-checking capabilities of an SMT solver [Lin & Mil+ 16]. The CSE implementation is compared to conventional SE, showing that incremental solving is more beneficial for exhaustive exploration than non-incremental solving [Lin & Mil+ 16]. To fully benefit from incremental solving, the search heuristic must be confined to depth-first search (DFS), a drawback that must not be underestimated according to [Lin & Mil+ 16].

While incremental symbolic execution techniques are not suitable for targeted search, which is subject to a series of research [Cad & Dun+ 08; San & Har 10; Ma & Kho+ 11; Böh & Pha+ 17], they are beneficial in reducing the cost for multiple SEs of a program during software testing [Yan & Khu+ 13].

Memoise is a technique that records and reuses results of previous applications of SE to a program across different runs by relying on an efficient tree-based data structure [Yan & Khu+ 13]. This *trie* is a compact representation of the visited symbolic paths during SE and additionally stores the branching decisions and the corresponding symbolic values [Yan & Khu+ 13]. An already executed path can be efficiently replayed by turning off constraint solving and guiding the search along the stored choices in the trie [Yan & Khu+ 13]. The preliminary evaluation has shown that the iterative deepening and directed exploration using the trie-based

data structure significantly reduced the cost for multiple SEs on a program during software testing. MEMOISE can also be used for cost-effective regression analysis by executing SE only on paths leading to change-impacted trie nodes and pruning others from the search [Yan & Khu+ 13].

Another form of memoization is the reuse of function summaries. For instance, the function summaries generated by SMART were used to revalidate them on the new program version [God & Lah+ 11]. A three-phased algorithm was presented, which solves the problem of statically validating symbolic test summaries through the use of SA and VCG. The algorithm decides by static checking whether the summary is still valid in the new program version or has to be dynamically recomputed [God & Lah+ 11]. The evaluation has shown that in the presence of minor code changes, this algorithm significantly improves the regression testing (RT) process [God & Lah+ 11].

In the subsequent section, the problem of creating new test cases that specifically target the reconfigured behavior in the new program version is addressed [San & Chi+ 08].

3.2 Test Suite Augmentation via Regression Analysis

Regression analysis aims to ensure that reconfigurations to a program P do not introduce any regressions in the reconfigured program P'. This is usually done by providing the same input to both program versions and comparing their outputs, as depicted in Figure 3.2.

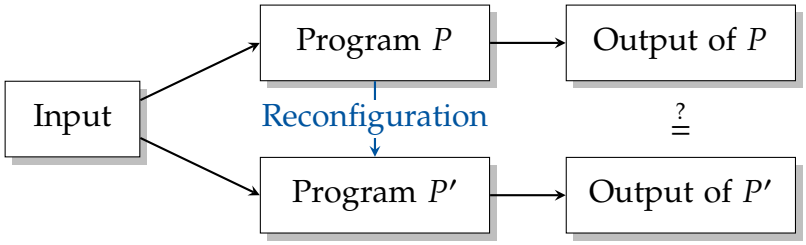


Figure 3.2: General overview of the regression analysis after a reconfiguration.

Regression analysis can be subdivided into regression verification and RT. The term regression verification was coined in [God & Str 09], in which an algorithm for an automatic and incremental proof of equivalence between two closely related versions of a program was implemented. The equivalence proof is based on the partial equivalence rule derived from Hoare's inference rule for recursive invocations [God & Str 09]. It either proves the absence of regressions by establishing behavioral equivalence or provides a witness of behavioral difference. Regression Verification Tool (RVT) traverses the two call-graphs of the loop-free and recursion-free fragments of the two programs and abstracts callees with uninterpreted functions when possible. This bottom-up decomposition algorithm

proved the partial equivalence of pairs of functions and was successfully evaluated on several small industrial programs within a reasonable time. Nevertheless, in practice, regression verification of non-equivalent versions is a very time and memory-consuming task [God & Str 09].

While regression verification tries to prove that both program versions behave equally or differently in a formally specified way [Fel & Gre+ 14], RT is usually concerned with reusing an existing test suite and re-running it on the reconfigured program. However, re-running the test suite of the old program version on the new one might be infeasible if it is too large and might not test the changed behavior [Böh & Roy+ 13; Xu & Kim+ 15]. The former is commonly addressed by RT techniques such as test case selection or prioritization, while the latter is a complementary techniques called TSA as illustrated in Figure 1.12 of Section 1.3.1.

The following sections will focus on state-of-the-art techniques with regard to TSA, as the algorithm presented in Chapter 5 focuses on generating test cases that exhibit divergent behavior in reconfigured PLC programs.

3.2.1 Program Differencing using Summarization

Differential symbolic execution (DSE) applies SE to perform a differential program analysis to create deltas that can be used for a variety of subsequent analyses [Per & Dwy+ 08]. The algorithm consists of several steps. First, the syntactic similarities are determined by a line-by-line comparison between both program versions. Next, for the methods that differ, both versions of those methods are symbolically executed, and summaries are generated. The SE over-approximates the behavior of the two program versions by constructing symbolic function summaries. The symbolic function summaries consist of a set of disjoint conditions on the input values, each of which is assigned the corresponding symbolic output value of the program [Per & Dwy+ 08]. Uninterpreted functions are used for the previously identified common code fragments to abstract from the concrete behavior and facilitate the analysis. The generated summaries are checked using two different notions of equivalence: *functional* and *partition-effects* equivalence. The functional equivalence checks whether, for the same input, the same output is emitted similarly to [God & Str 09]. The partition-effects equivalence checks for functional equivalence and whether the programs equivalently partition the input space, i.e., for the same inputs, the same outputs are emitted while traversing the same paths. If those checks fail, the summaries are not equivalent. DSE will then generate deltas that precisely characterize the input valuations and conditions under which the two program versions show divergent behavior.

While DSE is a precise analysis, it does not scale well. The main result of [Pod & Cla 90] shows that there is a relation between syntactic and semantic dependencies and that dependency analysis can give an "approximate" answer to the question if one statement is semantically dependent on another statement. SA avoids undecidability problems by giving up completeness for soundness, a property

which is exploited in the techniques used in [Per & Yan+ 11; Yan & Per+ 14] and presented next.

3.2.2 Aiding Regression Analysis with Change Impact Analysis

Directed incremental symbolic execution (DiSE) combines static change impact analysis (CIA) and state-of-the-art SE techniques to enable a more efficient regression analysis [Per & Yan+ 11]. The effect of the reconfiguration on the new program version is computed using a lightweight syntactical differential analysis. Afterward, DiSE begins performing an intraprocedural analysis to detect parts of the CFG affected by this reconfiguration using the information of the syntactical differential analysis. The computation of affected and unaffected vertices is based on control- and data flow dependencies and corresponds to some form of intraprocedural slicing. This information is then leveraged to perform a directed SE, which is guided toward those paths that can reach a reconfigured vertex. For any sequence of affected vertices that lie on some feasible execution path within the specified depth bound, DiSE explores one execution path containing that particular sequence [Per & Yan+ 11].

Several extensions and improvements have been researched that build up on the concept of DiSE. A natural extension of DiSE to interprocedural directed incremental symbolic execution (iDiSE) was proposed in [Run & Per+ 12] in which the intraprocedural analysis was lifted to an interprocedural analysis using call graphs to compute the impact of the reconfiguration across function boundaries.

Later, another extension of iDiSE to concurrent programs was presented as Conci-SE [Guo & Kus+ 16], in which concurrent paths in a program are analyzed using a summary-based incremental SE. At the core, Conci-SE relies on the static CIA of iDiSE. It computes symbolic summaries to infer execution paths affected by a reconfiguration in order to execute those affected paths incrementally. The concepts of DiSE were also used to improve scalability in regression verification [Bac & Per+ 13] by removing code fragments unaffected by the reconfiguration before analysis. It was then tried to show equivalence on the reduced programs [Bac & Per+ 13], and the evaluation concluded that this reduction technique is beneficial in lowering solving time.

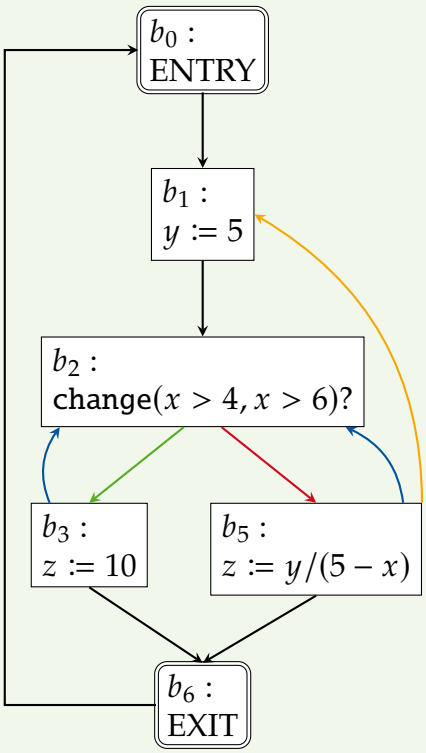
While the results of the DiSE analysis can be used for various software evolution tasks [Per & Yan+ 11], such as in regression verification [Bac & Per+ 13] or in RT [Yan & Per+ 14], it nonetheless suffers from two severe disadvantages. The static CIA gives only a conservative estimate of the impact of the reconfiguration. While this may lead to significant savings in analysis time [Yan & Per+ 14], it is too imprecise for most reconfigurations and especially not suitable for the analysis of reconfigurable PLC software, as explained in Example 3.1.

Example 3.1: Degeneration of Static Change Impact Analysis

This example emphasizes why a static CIA, often found in literature, is in most cases not suitable to analyze PLC programs.

```
1  PROGRAM P
2  VAR_INPUT
3      x : INT;
4  END_VAR;
5  VAR
6      y : INT;
7  END_VAR;
8  VAR_OUTPUT
9      z : INT;
10 END_VAR;
11 y := 5;
12 IF change(x > 4, x > 6)
       THEN
13     z := 10;
14 ELSE
15     // DIV BY ZERO ?
16     z := y / (5 - x );
17 END_IF;
18 END_PROGRAM;
```

Listing 3.1: Example POU.



Listing 3.2: CFG of example POU.

The static CIA of DiSE would mark all vertices of the CFG as affected by the reconfiguration by following the control- and data dependence edges, colored blue and orange, respectively. Performing SE on this "sliced" program corresponds to performing SE on the whole program. Furthermore, DiSE would yield an arbitrary value satisfying $\neg(x > 6)$, which does not guarantee that a potential divergence is revealed. Hence, static CIA degenerates for PLC software as it is often too imprecise and misses important pruning and prioritization opportunities, such as in a dynamic analysis.

A more expensive semantic analysis can be used to overcome the imprecision of the static CIA and retrieve more refined results. Similar to DSE, SymDiff tries to solve the problem of differential verification [Lah & Haw+ 12]. In more recent publications [Gyo & Lah+ 16; Gyo & Lah+ 17], SymDiff was extended by an interprocedural semantic CIA using equivalence relations which showed significant improvements in the precision of the analysis of larger projects. While SymDiff can prune the space of execution paths for which regression tests need to be generated [Gyo & Lah+ 16], it does not automatically derive test cases that

trigger divergent behavior.

To conclude, DiSE does not guide the SE in the direction of real divergence besides the reachability of the reconfigured vertices. It also explores one execution path at a time containing affected vertices within a specified bound which might not propagate a divergent behavior across multiple cycles in a PLC software setting. It is becoming apparent that techniques are needed that partition the input space with regard to failure [Wey & Jen 91], i.e., either *equivalence-revealing* or *difference-revealing*, to be able to propagate divergent behavior to the observable outputs of the PLC software.

Partition-based regression verification (PRV) is a technique that infers differential input partitions that represent a subset of the common input space of two program versions [Böh & dS O$^+$ 13]. It uses random testing and concolic execution to compute partitions characterized by symbolic conditions. The symbolic conditions define a range of valid inputs for which the specific partition [Böh & dS O$^+$ 13] propagates the same differential state to the output. If both versions compute the same output for an arbitrary input of the respective partition, the partition is said to be equivalence-revealing. It is then soundly guaranteed [Böh & dS O$^+$ 13] that both programs compute the same output for all inputs satisfying the symbolic condition of that partition. Otherwise, the respective partition is said to be difference-revealing. Additionally, PRV is able to generate test cases that expose different behavior across both program versions [Böh & dS O$^+$ 13]. A strength of PRV is the verification of entire input space partitions and the capability of relating behavioral differences to syntactic changes as opposed to DiSE [Per & Yan$^+$ 11] by using static and dynamic slicing during inference of the differential partitions. Another vital property of PRV, in contrast to traditional regression verification, lies in its incremental nature. PRV gradually verifies the differential partitions and hence can retain partial verification guarantees of already explored differential partitions even in case of interruption due to time or memory constraints [Böh & dS O$^+$ 13]. Example 3.2 illustrates the inferred differential partitions by PRV for the program shown in Example 3.1.

Example 3.2: Differential Partitions

Tables 3.1 and 3.2 show the partitioning of the partition-based regression verification on the running example shown in Example 3.1.

Table 3.1: Table with partitions.

	Input	Output
P	$x > 4$	$z := 10$
	$x \leq 4$	$z := 5/(5-x)$
P'	$x > 6$	$z := 10$
	$x \leq 6$	$z := 5/(5-x)$

The algorithm tries to partition the behavior of the programs by exploration of all combinations using the following relation:

$$s_1 \otimes s_2 = \{\big((c_1 \cap c_2), (o_1 \stackrel{!}{=} o_2)\big) \mid (c_1, o_1) \in s_1 \wedge (c_2, o_2) \in s_2 \wedge (c_1 \cap c_2) \neq \emptyset\} \ ,$$

where

$$o_1 \stackrel{!}{=} o_2 := \begin{cases} EQ, & \text{if } o_1 = o_2 \\ (o_1, o_2) & \text{otherwise.} \end{cases}$$

For the illustrated partitions in Table 3.1 the following combinations result when applying the relation:

$$\begin{aligned} (x > 4, z := 10) \otimes (x > 6, z := 10) &= \big((x > 4 \cap x > 6), EQ\big) \\ &= (x > 6, EQ) \\ (x > 4, z := 10) \otimes (x \leq 6, z := 5/(5-x)) &= \big(5 \leq x \leq 6, \\ &\qquad (z := 10, z := 5/(5-x))\big) \\ (x \leq 4, z := 5/(5-x)) \otimes (x > 6, z := 10) &= (\emptyset, \ldots) \\ (x \leq 4, z := 5/(5-x)) \otimes (x \leq 6, z := 5/(5-x)) &= \big(x \leq 4, EQ\big) \end{aligned}$$

The results are depicted in Table 3.2.

Table 3.2: Table with the resulting differential partitions.

	Input	Output
PRV	$x > 6$	EQ
	$5 \leq x \leq 6$	$z := 10 \wedge z := 5/(5-x)$
	$x \leq 4$	EQ

While $x > 6$ and $x \leq 4$ are differential partitions, i.e., a common subset of the input space of P and P', they propagate the same differential state

> to the output and hence are denoted as equivalence-revealing (EQ). Both programs lead to different outputs only for values satisfying $5 \leq x \leq 6$ and consequently may be difference-revealing.

Unlike DiSE, PRV can classify and derive test cases that lead to divergent outputs in both program versions. While PRV is a valuable alternative to other regression test generation techniques due to the derivation of homogeneous differential partitions concerning the failure [Wey & Jen 91], it does not scale well. The verification of the entire input space can be prohibitively expensive and is often not required in RT.

In the next section, a possible improvement is presented, which uses a test case from the previous program version as a "seed" to derive a test case for the new program version that might be difference-revealing.

3.2.3 Exposing Divergent Behaviors after a Reconfiguration

SSE is a technique for the derivation of difference-revealing test cases [Pal & Kuc+ 16; Kuc & Pal+ 18]. The goal is not to generate a complete test suite for the new program version but to derive test cases in which both programs expose divergent behavior. These test cases can be used to augment the test suite, which then is checked by a developer to determine whether the new behavior due to the reconfiguration is intended or not.

In contrast to CONCI-SE, which first symbolically executes the old version of the program to obtain the summaries [Guo & Kus+ 16] or PRV, which runs both program versions separately, SSE executes both versions at the same time. The similarity of the two analyzed programs can be exploited by sharing the symbolic store and reducing the required memory.

As input, SSE receives a CAP and a test suite for the program version before reconfiguration. The CAP is obtained by annotating the old program version with a `change`-annotation macro "`change(old:` e_1`,new:` e_2`)`" such that the old version and new version are obtained by replacing all occurrences of change with the respective old or new argument (see Section 2.5).

SSE then proceeds to select those test cases that "touch" the patch, i.e., the vertices of the CFG, which are reconfigured [Pal & Kuc+ 16]. Afterward, a CE is performed on the CAP to derive divergent contexts. A shadow expression containing information about both program versions is generated upon encountering a change expression. The shadow expression symbolically represents an expression for both program versions, which prevents the duplication of common expressions and the creation of two separate memory stores for each version.

If a branch is reached, execution might diverge. For this purpose, SSE checks whether the current expression contains any shadow expression that makes the two versions follow different paths. While the executions might follow the same

path under the current valuations, SSE also checks whether input valuations exist that might make both versions take different paths. A test case is then generated for each possible input valuation and added to the queue of divergent contexts. In case different paths are followed, execution is stopped, and the current execution context is added to a queue of divergent contexts. Then, a bounded symbolic execution (BSE) is performed on the prior derived divergent contexts and corresponding test cases to explore potential divergent behavior in the new program version. Finally, both versions are executed on all derived divergent inputs, and the results are presented to the developer.

SSE is driven by the concrete inputs from an existing test suite, so visiting a reconfigured vertex in the CFG is trivially necessary to exercise the reconfiguration. This also implies that divergences might be missed, as this technique strongly depends on the quality of the initial valuations. Compared to PRV, the exploration strategy of SSE focuses on constraining the search space by inheriting the path constraint prefix from the concrete input valuations that reach the potential divergence. As far as PLC software is concerned, this might lead to missing more profound nested divergences in the breadth-first search phase of SSE, as divergent behavior might need to be propagated across multiple cycles. The idea of SSE is adapted and extended in [Nol & Ngu+ 19] to "Complete SSE" by exhaustively exploring the execution tree of the reconfigured program. However, this thesis solely focuses on evaluating the feasibility of the SSE approach.

3.3 Related Work

An overview of the challenges for the functional testing of reconfigurable CPPSs is given in [Zel & Wey 16]. [Zel & Wey 16] identifies three use cases in which the requirements of future production systems have several implications on the need for functional testing. These challenges arise through changing the production environment, different production tasks, and the system's decentralization. This leads to reconfigurations of software and hardware, the often changing and ad hoc production tasks, and the possibility of multiple production paths caused by resource redundancy [Zel & Wey 16].

Software components must be adequately tested and verified throughout the whole life cycle of the CPPS to ensure correct behavior after a reconfiguration [Gro & Sim+ 20]. In addition to the two mutually complementing approaches presented in [Gro & Sim+ 20], this section aims to give an overview of other relevant approaches in the field of verification and testing of PLC software. Table 3.3 gives an overview of related work regarding the verification and testing of PLC software used in CPPSs. The publications and techniques in Table 3.3 are categorized according to their field of application, i.e., level in the development cycle, their view, and whether they target software reconfigurations. They are subdivided into verification and testing, and the "! " symbol denotes that the techniques, while not intended in the respective publication, can be lifted to perform the necessary task

Table 3.3: Overview of the related work in the field of PLC software verification and testing.

Source	Level [VDI 21]	View	Structural	Functional	Reconfigurations
Verification					
ARCADE.PLC – CYCLE-BMC [Boh & Ham+ 18]	Unit / Integration	Inner	!	✓	✗
VERIFAPS [Bec & Ulb+ 15; Wei & Ulb+ 20] – Regression Verification	Unit / Integration	Inner	✗	✓	✓
– GTT and RTT [Wei & Wie+ 17; Wei 21]		Inner	✗	✓	✓
TESTIAS [Zel & Jaz+ 18; Zel & Wey 18; Zel & Jaz+ 19]	HW/SW Integration	Outer	✗	✓	✓
Testing					
TWISTTURN [Thö & Rei+ 17; Thö 21]	HW/SW Integration	Outer	✗	✓	✗
Regression Testing with CoDeSys [Ule & Vog+ 17; Ule & Vog 18; Ule 18]	HW/SW Integration	Inner	!	✓	!
ARCADE.PLC – TSG [Sim & Fri+ 15; Boh & Sim+ 16; Sim & Kow 18]	Unit / Integration	Inner	✓	✗	✗
CONTRIBUTION OF THIS THESIS – TSG + TSA	Unit / Integration	Inner	✓	✗	✓

of the individual category.

3.3.1 Verification of Programmable Logic Control Software

The verification of PLC software is subject to a lot of research not limited to academia [Adi & Dar+ 15; Dar & Bla+ 15; Lop & Tou+ 22]. PLCVERIF is a tool developed at the European Organization for Nuclear Research (CERN). It features a software verification pipeline commonly found in other works such as ARCADE.PLC [Boh 21] in which a PLC program is transformed into an IR (see Section 2.2), and verification tasks are delegated to state-of-the-art model checkers with requirements expressed in some form of temporal logic such as Computation Tree Logic (CTL) [Bai & Kat 08]. The following sections discuss and relate the related work's general ideas to this thesis' contributions.

VERIFAPS

A significant bottleneck in applying formal methods is the lack of appropriate system and requirement specifications [Bec & Mun+ 19]. Regression verification circumvents this problem by reusing the version of the PLC software before the reconfiguration, thus reducing the need to formalize requirements [Bec & Mun+ 19]. VERIFAPS is a software project for verifying automated production plants in which such methods are explored.

Regression Verification [Bec & Ulb+ 15] proposed a novel method for regression verification of PLC software. The novelty of the contribution is the definition of reactive conditional and relational equivalence and a proof methodology for PLC software.

For this purpose, the trace semantics and trace equivalence of PLC programs are introduced for the formalization of the relational equivalence [Bec & Ulb+ 15]. Furthermore, an environment model can be used to increase the precision of the verification task. VerifAps accepts PLC programs in ST and transforms them into an internal representation ST0, in which the program is fully unwound, and invocations are inlined.

VCGs are obtained using a SE to derive a state transition system, which serves as an input for a model checker. Full equivalence can be restricted using additional constraints to allow deviations due to reconfigurations representing intended behavior in the new program version [Bec & Ulb+ 15]. This shares the workload between the tool-assisted invariant generation and the user by encoding both the old and the new program version via coupling invariants [Bec & Ulb+ 15].

A successful evaluation of the PPU benchmark demonstrates the feasibility of the proposed technique for practical evolution scenarios and uncovers a series of regression bugs. Albeit slow, the results are promising.

Regression verification provides an equivalence proof for all possible inputs, increasing confidence in the correctness of the reconfigured PLC software. It does not require any functional or behavioral specification as the code of the prior version is used. While proofs can be harder to obtain when the PLC software increases in size, modularization tries to mitigate this problem which was investigated in further research [Wei & Ulb+ 20; Wei 21].

Gtt and Rtt Generalized test tables (Gtt) are a new specification language for defining test cases of functional properties used for verification [Wei & Wie+ 17]. The typical definition of a test case as a single table is lifted to capture multiple similar behavioral cases via a formal temporal specification.

The table consists of four significant columns. The first column denotes the test step, whereas the second and third column represents a set of input and output variables. The fourth column represents the duration of the respective step. Each input and output variable contains constraints on the respective variable. The rows are applied consecutively from top to bottom [Wei & Wie+ 17]. Depending on the given constraints in the duration column, a row or a group of rows can be skipped or repeated [Wei & Wie+ 17].

The test conformance of a Gtt is defined as a two-party game over infinite words between the system under test (SUT) and the tester [Bec & Cha+ 17]. The game is encoded into a model which can be efficiently validated by state-of-the-art model checkers [Bec & Cha+ 17], and the applicability and feasibility were demonstrated in several experiments. Furthermore, an empirical study evaluated the usability and comprehensibility of the added syntactical Gtt notations concerning table

generalization [Wei & Wie+ 17].

The concept of GTTS was extended to relational test tables (RTTS) in [Wei 21]. As the name suggests, RTTS is an extension of GTT for specifying relational properties. A functional property is specified over the behavior of a single program run, whereas a relational property is specified over multiple runs of a program [Wei 21]. The semantics are defined as a reduction to GTT by supplying an instrumented product program to the GTT verification engine [Wei 21]. This allows RTT to use the existing program version as a functional specification. Only the relation between the subsequent program versions have to be defined using a relational specification to ensure that unwanted regressions were not introduced during the reconfiguration [Wei 21].

TESTIAS

TESTIAS is a tool for verifying reconfigurable distributed control systems [Zel & Jaz+ 19]. The underlying concept of component-based verification was researched in several publications [Zel & Jaz+ 18; Zel & Wey 18]. The mode of operation of verifying a reconfiguration is divided into three tasks.

At first, an impact analysis on the dependency graph of a block definition diagram of the automation system for identifying affected components is performed. This static CIA determines which components are affected by traversing the edges representing a call to a component's functionality. Next, a composition of the behavior models of the affected components relevant to the requirements under verification is carried out. Adapted Petri nets are used as the underlying behavior model for the composition, which abstracts and encapsulates the internal behavior of the component and focus on the representation of the interfaces.

The composition thus works on the level of interfaces of the adaptated Petri nets and is able to verify interoperability [Zel & Wey 18]. Last, the affected subsystem is verified using a state-of-the-art verification tool. The evaluation of TESTIAS comprises several reconfiguration scenarios and requirement changes, showing the technique's feasibility.

CYCLE-BMC

CYCLE-BMC is a lifted variant of BMC to the domain of PLC software for checking the reachability of unwanted behavior within a bounded number of steps [Boh & Ham+ 18]. In order to achieve an efficient, incremental, and monotonic logical characterization of the PLC software's semantics, a combination of SE and dynamic large-block encoding (LBE) is used [Boh & Ham+ 18].

Parts of this encoding are used in this thesis to generate summaries of FBs. The PLC program under analysis is symbolically encoded in the topological order induced by the underlying control-flow automaton (CFA). Using assumption literals and solving under assumptions, the incremental solving capabilities of the underlying SMT solver are exploited. The logical characterization is limited to

inlined PLC programs. The evaluation shows that an analysis specially tailored to the domain of PLC programs leads to significant improvements over related work [Boh & Ham+ 18].

3.3.2 Testing of Programmable Logic Control Software

TwistTurn

Hardware-in-the-loop (HIL) simulation is a powerful technique to reduce the risk of system failures leading to expensive damage [Thö & Rei+ 17]. The SUT is connected to the HIL simulator and stimulated by test cases defined in a language extension of ST. The output behavior of the SUT is analyzed with regard to user-defined acceptance criteria.

In this testing scenario, a PLC is used to control an industrial plant simulated using TwistTurn. This concept was lifted in [Thö & Sma+ 19], in which a randomized test case generation framework was used. The test cases were generated using fuzzing and executed via OPC Unified Architecture (OPC UA) using an "ioco"-based analysis. OPC UA is a cross-platform standard for the exchange of data between components in a SOA developed by the OPC Foundation [IEC 20]. An exhaustive evaluation using TwistTurn as a testing tool is available in [Thö 21].

Regression Testing with CoDeSys

In [Ule & Vog 16], RT for reconfigurable PLC software was researched. RT is performed on the system level by re-evaluating functional test cases relating to the whole CPPS and leveraging field data to build a correlation between the SUT and the executed test suite [Ule & Vog 16].

The relation of system tests concerning the tested components is done in a guided, semi-automatic approach in which a human operator is included to perform the manual tasks [Ule & Vog 16]. This information is used to prioritize test cases that are modification traversing in subsequent program versions. The concept was expanded and deepened in further work [Ule & Vog 18], and further research has looked at a coverage assessment approaches using behavior models [Ule & Vog+ 17].

In comparison to this thesis, untested behavior is identified on the system level based on requirements [Ule & Vog 18]. However, the contribution of this thesis focuses on generating test cases on the unit level using structural test coverage criteria.

Arcade.PLC

[Sim & Fri+ 15] presents a method for automatic TSG for PLC software using coverage metrics. A model checker is used to generate line or branch coverage-based

test cases by iteratively deriving counterexamples. The counterexamples serve as witnesses of reachability, and the input variables' valuations are transformed into test cases that can be executed on the actual hardware.

The algorithms are implemented in the tool ARCADE.PLC and were evaluated on a vendor-specific implementation of the PLCopen Safety suite. A refined and more efficient approach targeting TSG for PLC software is presented in [Boh & Sim+ 16]. A concolic testing strategy is employed to substitute the expensive model-checking in case unreachable branches exist. Another heuristic for the reduction of the analysis time in TSG¸ is the exploitation of mode spaces of a PLC program [Sim & Kow 18].

Concluding Remarks

While there is work that focuses on the generation of test cases about the functional perspective of PLC software, there is no work that tries to generate and augment a test suite after a reconfiguration to the structural level. This thesis aims to fill this research gap by improving the scalability of TSG and investigating the applicability of SSE in the domain of PLC software for TSA.

4 Test Suite Generation

This chapter focuses on the algorithms for TSG. In the following sections, the methodology and implementation of a BSE are explained in detail, and the design space choices of Section 2.4 are discussed. Figure 4.1 shows the integration of TSG

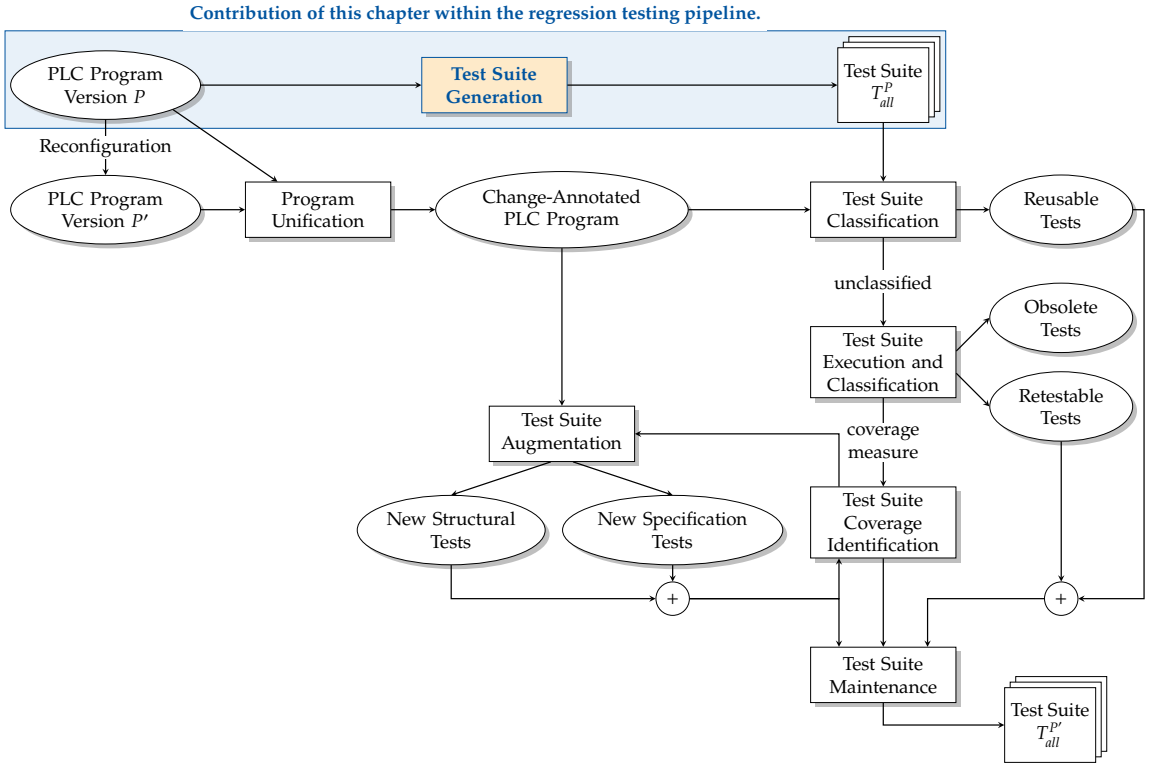
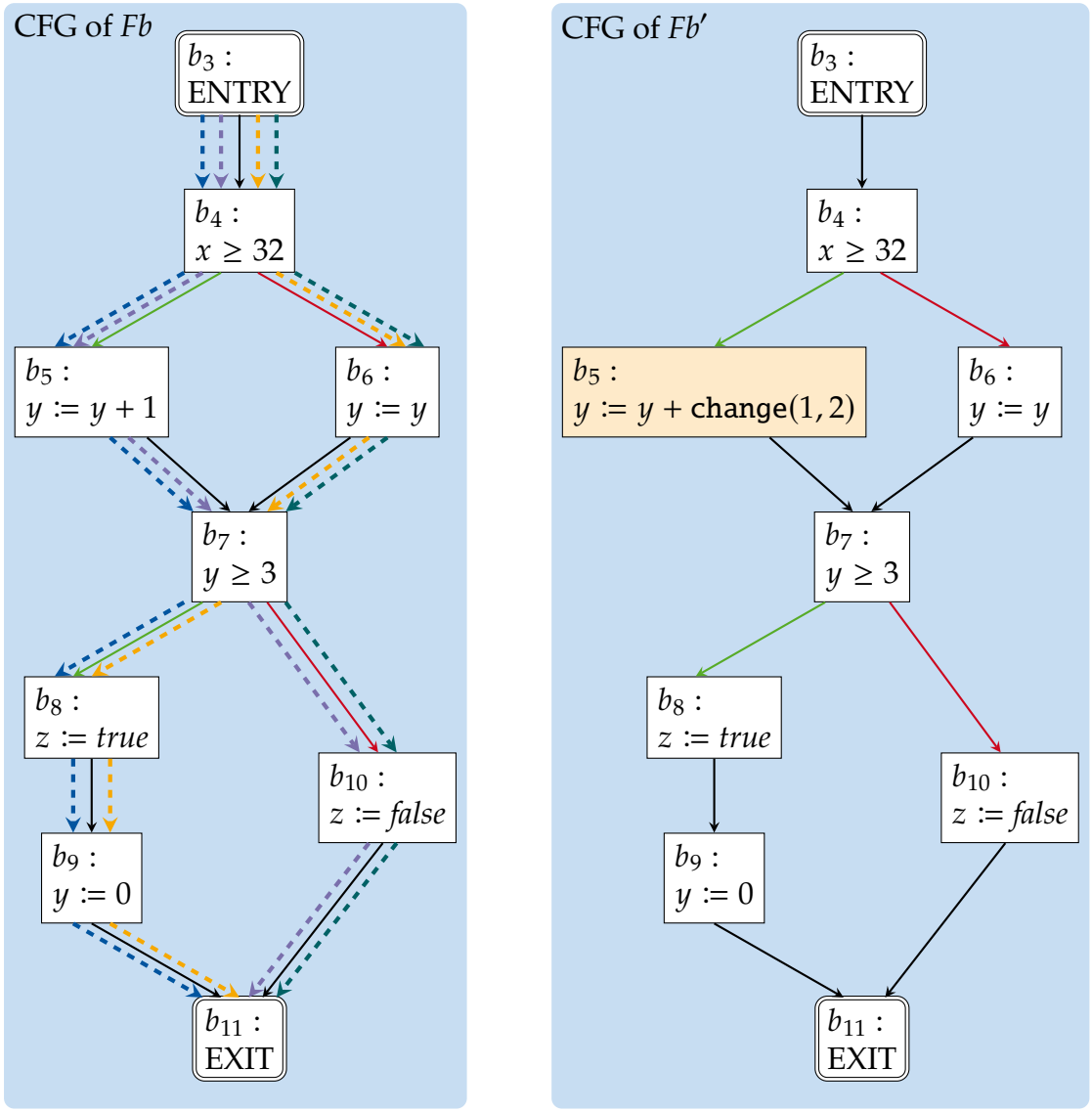


Figure 4.1: Overview of the regression testing pipeline and contribution of this chapter.
Figure adapted from Figure 3 in [Leu & Whi 89].

into the overall regression testing and software maintenance pipeline. Not only the resulting test suite T^P_{all} but also the main algorithm driving the TSG are essential components for the algorithms in the next Chapter 5 and thus explained in detail. Furthermore, this chapter introduces the concept of summarization to increase scalability concerning compositionality by using symbolic summaries.

Running Example The following reconfigured FB should serve as a running example throughout this chapter.



(a) CFG of the old program version with annotated summaries.

(b) CFG of the reconfigured program for which summaries should be checked.

Figure 4.2: CFGs of the old and the reconfigured program versions.

4.1 Compositional and Bounded Symbolic Execution

One goal of SE is to drive the execution of a program along certain paths of interest to maximize a coverage criterion. Throughout this thesis, the acronyms SE, CSE, and BSE are used interchangeably to abstract from the parametrizable algorithms for the particular problem instance within the PAF. Frames are used to analyze PLC programs with functions and FBs and are essential to provide some sense

of context-sensitivity in compositional analysis. Each frame carries the current *scope* such that the variables are always uniquely referencable, which goes hand in hand with the chosen IR, as discussed in Section 2.3. Another vital component of a frame is the return vertex. It is mandatory to ensure that only realizable paths are explored, which is in detail explained in Section 4.1.1.

Definition 4.1: Frame

A *frame* f is a tuple $f = (G, scope, b_{\ell_{return}})$, where

- G represents the CFG of the POU under analysis,
- *scope* is the scope under which references to variables should be evaluated,
- $b_{\ell_{return}}$ denotes the vertex in the caller's scope to which execution should return after reaching the respective exit vertex of the callee.

Given Definition 2.10 of an execution state and Definition 4.1 of a frame, the primary data structure for BSE is introduced next.

Definition 4.2: Execution Context

An *execution context* q is a tuple $q = (c, s, C)$, where

- $c \in \mathbb{N}$ denotes the current execution cycle of the PLC program,
- $s = (b_\ell, \rho, \sigma, \pi)$ is an execution state,
- C is a call stack carrying frames.

The execution context encapsulates the current execution state and serves as the internal representation of the BSE's state. Execution of a PLC program is implemented by appropriate modification of the execution context q in the PAF. Algorithm 1 illustrates BSE's method of operation.

Algorithm 1 Bounded Symbolic Execution

Input : Program $P = (G, \mathcal{G})$
Local : Execution context q
Output : Test suite T

1: $q \leftarrow$ `initialize`(G)
2: **while** ¬`isGlobalTerminationCriteriaMet`() **do**
3: | // Algorithm 2
4: | $(q', tc_{local}) \leftarrow$ `executeCycle`(q)
5: | **if** tc_{local} **then**
6: | | terminate
7: | **end**
8: | $cycle \leftarrow cycle + 1$

9: **end**
10: **return** T

BSE receives the compiled program in the IR and generates a test suite T adhering to a selected coverage criterion.

Initialization An initial execution context q is derived from the main CFG of the program P in Line 1 of Algorithm 1. The current execution cycle c is initialized with 0, and the execution begins at the entry vertex b_{ℓ_e} of the main CFG. The concrete and symbolic stores are initialized regarding their data and storage types. Fresh symbolic variables are introduced for whole-program inputs, i.e., inputs belonging to the main CFG. All other inputs and variables were flattened during compilation and therefore are not initially treated symbolically. The call stack C is initialized with the main frame $\langle G, \text{getScope}(G), b_{\ell_e} \rangle$.

After initialization, the BSE is executed until the global termination criteria in Line 2 of Algorithm 1 are reached. As the execution of PLC programs is ad infinitum, the termination criteria can be either configured to stop the execution after a predefined bound, execute as long as a certain time has passed, the coverage criterion has been reached, or one or more of the prior criteria in combination has occurred. As long as the BSE should not terminate, the CFG of the program P is executed on a per-cycle basis in Line 4 of Algorithm 1.

Executing Cycles In order to account for the cyclic execution mode of PLC programs, BSE is able to execute the CFG on a per-cycle basis.

Algorithm 2 executeCycle

Input : Execution context $q_{in} \coloneqq (c, s = (b_\ell, \rho, \sigma, \pi), C)$
Local : Current execution context q, Priority queue Q, Merge queue $\mathcal{M}$, Test suite T
Output : Succeeding execution context q', local termination criteria tc_{local}

1: Q.push(q_{in})
2: **while** ($Q \neq \emptyset \vee \mathcal{M} \neq \emptyset$) **do**
3: | **if** `isLocalTerminationCriteriaMet`() **then**
4: | | **return** $(\_, tc_{local})$
5: | **end**
6: | **if** $Q = \emptyset$ **then**
7: | | Q.push(`merge`($\mathcal{M}$))
8: | **end**
9: | $q \leftarrow Q$.pop()
10: | // Algorithm 3
11: | $Q' \coloneqq \{q'_1, \ldots, q'_n\} \leftarrow$ `executeVertex`(q)
12: | **for each** $q' \coloneqq (c', s', C') \in Q'$ **do**
13: | | **if** `hasCoverageIncreased`(q') **then**
14: | | | T.push(`deriveTestCase`(q'))

```
15:         end
16:         if reachedSucceedingCycle(q′) then
17:             return (q′, _)
18:         else
19:             if reachedMergePoint(q′) then
20:                 M.push(q′)
21:             else
22:                 Q.push(q′)
23:             end
24:         end
25:     end
26: end
```

Algorithm 2 shows how execution is guided through a cycle. A cycle is executed as long as either execution contexts exist for exploration or execution contexts are waiting to be merged (see Line 2 of Algorithm 2). In order to guarantee the progress of BSE, Lines 3 to 4 of Algorithm 2 check whether a local termination criterion is fulfilled. In case BSE runs into a timeout or the wanted coverage has been reached, execution of the cycle is stopped. If the exploration queue Q does not contain any execution contexts, the engine performs a merge of the enqueued execution contexts at the respective merge points in Lines 6 to 8. Before continuing with the rest of Algorithm 2, the merge strategy is explained next.

4.1.1 Merge Strategy

Merging is an important technique to alleviate the path explosion problem mentioned in Section 2.4. During initialization in Line 1 of Algorithm 1, the queue of merge points $\mathcal{M}$ is populated with all possible vertices of the CFG at which control flow joins with respect to realizable paths.

Definition 4.3: Merge Point

A *merge point* mp is a tuple $mp = (scope, depth, b_\ell, b_{\ell_{return}})$, where

- $scope$ denotes the current calling scope,
- $depth$ denotes the call depth,
- b_ℓ denotes the vertex at which merging should occur,
- $b_{\ell_{return}}$ denotes the vertex to which the execution context in the current calling scope returns to.

A realizable path represents a valid interprocedural path, i.e., a path in which each interprocedural return edge corresponds to the preceding interprocedural call edge [Rep & Hor+ 95].

Example 4.1: Infeasible and Realizable Paths

Infeasible paths do not correspond to actual executions and must be considered when merging.

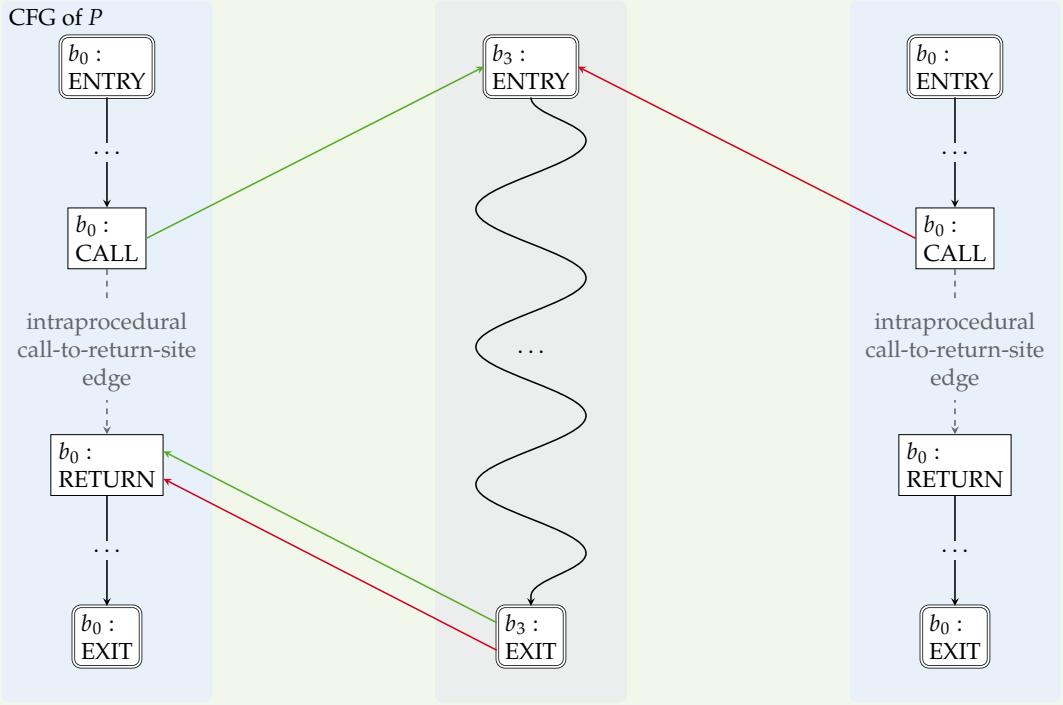


Figure 4.3: Two calls under different execution contexts to the same callee.

The realizable path is colored green, and the infeasible path is colored red. The definition of merge points prevents the merging of infeasible paths as it considers their calling contexts.

The merge strategy adheres to the representation of callees in the IR (see Section 2.2) and avoids merging of interprocedural invalid paths. It always prioritizes lower labeled vertices at the same depth.

Another strategy worth discussing is the exploration strategy hidden behind Line 9 of Algorithm 2.

4.1.2 Exploration Strategy

The execution follows a cycle-based and depth-first exploration strategy similar to prior work [Boh & Sim+ 16] with a parametrizable timeout, coverage, and cycle bound [Gro & Völ+ 22b]. The global termination criteria in Line 2 of Algorithm 1 feature a configurable cycle exploration bound that can be explicitly configured to the analyzed program to cope with increasing computation time induced by

the cyclic execution of PLC programs. This guarantees termination even in the presence of unreachable branches (discussed in Section 4.1.4).

Internally, a heuristically sorted priority queue Q is used, which determines the order of execution of the execution contexts q. When retrieving an execution context q from the priority queue Q, such as in Line 9 of Algorithm 2, execution contexts with a lower cycle count and deeper nested scopes are prioritized over "shallow" execution contexts. Ultimately, this results in executing all feasible execution paths through one execution cycle before continuing the execution in the next cycle [Gro & Völ+ 22b].

Another benefit of adapting the depth-first search heuristic to behave like a pseudo-breadth-first search through one execution cycle is the generation of concise test cases with no unnecessary executed cycles reaching the same vertices. Furthermore, the decision to prioritize execution contexts with deeper nested paths over the execution contexts with shallower nested paths ensures pushing the exploration frontier forward while giving rise to merge opportunities at control-flow joint points.

After retrieving the next execution context q from the exploration queue, it is executed in Line 11 of Algorithm 2.

Executing Vertices Depending on the type of vertex executed by the BSE, different modifications of the execution context q are performed.

Algorithm 3 executeVertex

Input : Execution context $q := (c, s = (b_\ell, \rho, \sigma, \pi), C)$
Output : Set of succeeding execution contexts $Q' := \{q'_1, \ldots, q'_n\}$

```
1:  switch handleVertex(b_ℓ) do
2:      case PROGRAM_ENTRY do
3:          b_ℓ' ← getSucceedingVertex(b_ℓ)
4:          s' ← (b_ℓ', ρ, σ, π)
5:          Q'.push(q' := (c, s', C))
6:      end
7:      case FUNCTION_BLOCK_ENTRY do
8:          // analogous to PROGRAM_ENTRY
9:          b_ℓ' ← getSucceedingVertex(b_ℓ)
10:         s' ← (b_ℓ', ρ, σ, π)
11:         Q'.push(q' := (c, s', C))
12:     end
13:     case REGULAR do
14:         // Algorithm 4
15:         Q' ← executeInstruction(q)
16:     end
17:     case PROGRAM_EXIT do
18:         ρ' ← ρ[v ↦ random() | v ∈ V_input]
19:         σ' ← σ[v ↦ v_fresh | v ∈ V_input]
```

```
20:       s' ← (b_ℓ', ρ', σ', π)
21:       Q'.push(q' := (c + 1, s', C))
22:     end
23:     case FUNCTION_BLOCK_EXIT do
24:       (_, _, b_ℓreturn) ← C.top()
25:       C' ← C.pop()
26:       s' ← (b_ℓreturn, ρ, σ, π)
27:       Q'.push(q' := (c, s', C'))
28:     end
29: end
30: return Q'
```

Upon encountering a program or FB entry vertex, the execution context is updated with the succeeding vertex induced by the underlying structure of the CFG.

The exit vertex of the main program or of an FB is handled uniquely. Upon encountering the program exit vertex, the execution context is prepared for the subsequent cycle by updating the concrete store with random valuations and introducing fresh symbolic variables for the symbolic store of the program input variables. The semantics follow the operational semantics defined in Appendix A, and the `random()` function assigns each input variable $v \in \boldsymbol{V}_{input}$ a type-specific random value $d \in \boldsymbol{D}$. In contrast, for each input variable in the symbolic store σ, a fresh symbolic input variable v_{fresh} is introduced.

Implicitly, valuations for the local and output variables are coupled to keep their concrete valuations from the previous cycle and reference the symbolic valuations from the prior cycle, if any. Furthermore, the cycle count is increased, leading to the termination of the local execution of the respective execution context through this cycle in the calling algorithm.

The handling of FB exit vertices differs from the program exit in that only the call stack C is modified, and the succeeding vertex is retrieved from the current frame. Do note that the handling of the call instruction is responsible for modifying the call stack C instead of handling the FB's entry vertex.

Before continuing with the handling of regular vertices in Line 15 of Algorithm 3, the Lines 12 to 24 of Algorithm 2 are discussed. Several checks are performed for each succeeding execution context q', which were derived from executing the vertex of the current execution context. In case the execution leads to an increase in coverage by, for example, the discovery of a yet uncovered vertex denoting a successor of a branching vertex, a test case is derived from the respective execution context and added to the test suite. Afterward, it is checked whether the execution context reached the next cycle, and execution is stopped if the beginning of a new cycle is reached. Concluding the discussion of Algorithm 2 with Lines 19 to 22, it is decided whether the resulting execution context reached a merge point and should be enqueued for merging or further exploration.

Next follows the remaining discussion of Algorithm 3 and Algorithm 4.

4.1.3 Assignments, Branches, and Calls

Algorithm 4 executes instructions under the current execution context q. The semantic effects of the instructions are captured by modification of the respective stores and the current execution context q.

Algorithm 4 executeInstruction

Input : Execution context $q \coloneqq (c, s = (b_\ell, \rho, \sigma, \pi), C)$
Local : Applicable summaries S
Output : Set of succeeding execution contexts $Q' \coloneqq \{q'_1, \ldots, q'_n\}$

1: **switch** `instructionAt`(b_ℓ) **do**
2: **case** $v \coloneqq e$ **do**
3: $b_{\ell'} \leftarrow$ `getSucceedingVertex`(b_ℓ)
4: $\rho' \leftarrow \rho[v \mapsto \text{eval}_\rho(e)]$
5: $\sigma' \leftarrow \sigma[v \mapsto \text{eval}_\sigma(e)]$
6: Q'.`push`$\big(q' \coloneqq (c, s' = (b_{\ell'}, \rho', \sigma', \pi), C)\big)$
7: **end**
8: **case** b **do**
9: $b_{\ell,true} \leftarrow$ `getSucceedingPositiveVertex`(b_ℓ)
10: $b_{\ell,false} \leftarrow$ `getSucceedingNegativeVertex`(b_ℓ)
11: **if** $\text{eval}_\rho(b)$ **then**
12: $\pi' \leftarrow \pi \wedge \text{eval}_\sigma(b)$
13: **if** `tryFork`$(\pi \wedge \text{eval}_\sigma(\neg b))$ **then**
14: $\rho' \leftarrow$ `model`$(\pi \wedge \text{eval}_\sigma(\neg b))$
15: Q'.`push`$\big(q'_{forked} \coloneqq (c, s' = (b_{\ell,false}, \rho', \sigma, \pi \wedge \text{eval}_\sigma(\neg b)), C)\big)$
16: **end**
17: Q'.`push`$\big(q' \coloneqq (c, s' = (b_{\ell,true}, \rho, \sigma, \pi'), C)\big)$
18: **else**
19: $\pi' \leftarrow \pi \wedge \text{eval}_\sigma(\neg b)$
20: **if** `tryFork`$(\pi \wedge \text{eval}_\sigma(b))$ **then**
21: $\rho' \leftarrow$ `model`$(\pi \wedge \text{eval}_\sigma(b))$
22: Q'.`push`$\big(q'_{forked} \coloneqq (c, s' = (b_{\ell,true}, \rho', \sigma, \pi \wedge \text{eval}_\sigma(b)), C)\big)$
23: **end**
24: Q'.`push`$\big(q' \coloneqq (c, s' = (b_{\ell,false}, \rho, \sigma, \pi'), C)\big)$
25: **end**
26: **end**
27: **case** $G()$ **do**
28: $b_{\ell_{return}} \leftarrow$ `getSucceedingIntraproceduralVertex`(b_ℓ)
29: C.`push`$(G', \text{`getScope`}(G'), b_{\ell_{return}})$
30: $S_{applicable} \leftarrow$ `findApplicableSummary`(q)
31: **for each** $s \in S_{applicable}$ **do**
32: $q' \leftarrow$ `applySummary`(q)
33: Q'.`push`(q')
34: **end**

```
35:   end
36: end
```

Assignments Lines 2 to 7 of Algorithm 4 are responsible for handling the assignment instruction. The succeeding vertex is determined from the underlying CFG, and the concrete and symbolic stores are updated via the respective evaluation function eval (see Definitions 2.7 to 2.8) concerning the written variable. The bracket notation denotes the usual replacement for the specified variable in the stores as defined in Section 2.3.

Branches Whenever a branching instruction is encountered, Lines 8 to 26 of Algorithm 4 analyze whether the control flow can branch in either the positive, the negative, or both directions. Depending on the concrete valuations of the current execution context q, Line 11 either evaluates to *true* or *false*. As the cases are mirrored and structurally analogous, the positive case is discussed in the following.

In case the condition of the branching instruction evaluates to *true*, the execution is continued with the positive branch, and the path constraint is updated symbolically. Next, Line 13 checks whether the other path would also be feasible under the current path constraint with the negated condition. If the underlying SMT solver returns a satisfying solution, the model can be retrieved to instantiate the concrete store, governing the concrete valuations that lead to this branch. A forked execution context is created and added to the set of succeeding execution contexts.

Calls Call instructions are lowered to a sequence of pre- and post-assignments (see Appendix A for further details). Therefore, Lines 27 to 35 are responsible for modifying the call frame stack and updating the control flow appropriately. While not adhering to a feasible execution, the underlying CFG exposes an intraprocedural control-flow edge that relates the calling vertex with the returning vertex (see Line 28 of Algorithm 4) in the calling CFG. This way, the frame carries context-sensitive information regarding the calling context, enabling the merge strategy only to consider feasible execution paths.

Example 4.2: Symbolic Execution through the Running Example

This example shows how test cases may be derived during SE. Consider the following execution context enters the FB of Figure 4.2a during SE.

$$\begin{array}{l} \pi : \mathit{true} \\ \rho : \{x \mapsto 32, y \mapsto 0, z \mapsto \mathit{false}\} \\ \sigma : \{x \mapsto x_0\} \end{array}$$

After the execution of the branch in Lines 8 to 26 of Algorithm 4, the set of succeeding execution contexts Q' holds the two execution contexts q_1 and q_2.

The resulting execution contexts are illustrated in Figure 4.4.

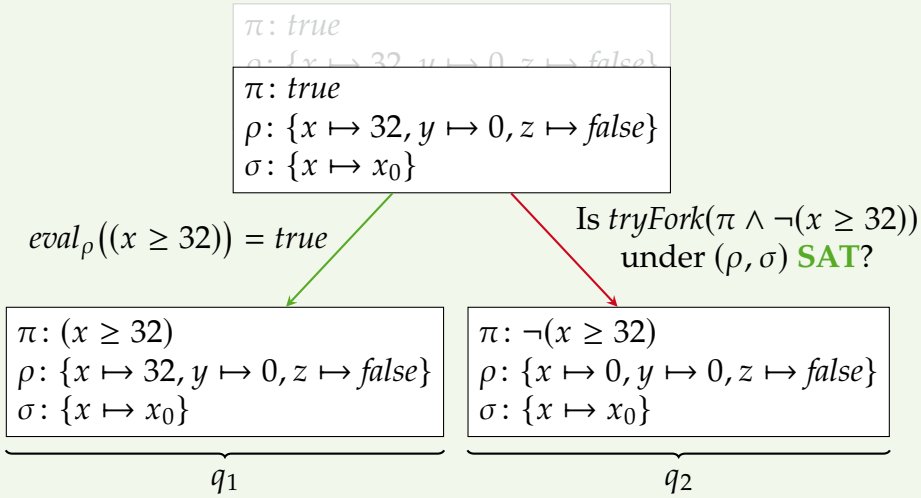


Figure 4.4: Intermediate state of the SE after a fork has occurred.

These execution contexts are propagated through Algorithm 3 to Algorithm 2, for which two test cases are generated in Lines 13 to 15 as both vertices were traversed for the first time, and hence the overall coverage has increased. The derived test cases consist of the respective input valuations in each cycle and the initial concrete state valuations. The resulting test cases are depicted in Figures 4.5 to 4.6 in the XML format used by the PAF.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <testcase>
3      <initialization>
4          <valuation variable="P.b">false</valuation>
5          <valuation variable="P.f.x">0</valuation>
6          <valuation variable="P.f.y">0</valuation>
7          <valuation variable="P.f.z">false</valuation>
8      </initialization>
9      <input cycle="0">
10         <valuation variable="P.a">32</valuation>
11     </input>
12 </testcase>
```

Figure 4.5: Resulting test case for execution context q_1.

```xml
<?xml version="1.0" encoding="utf-8"?>
<testcase>
    <initialization>
        <valuation variable="P.b">false</valuation>
        <valuation variable="P.f.x">0</valuation>
        <valuation variable="P.f.y">0</valuation>
        <valuation variable="P.f.z">false</valuation>
    </initialization>
    <input cycle="0">
        <valuation variable="P.a">0</valuation>
    </input>
</testcase>
```

Figure 4.6: Resulting test case for execution context q_2.

The test cases depict fully qualified variable identifiers of the running example shown in Figure 2.3. In order to obtain the corresponding state and output valuations, the test cases must be simulated using the semantics of the CFG under test until the end of the cycle is reached.

4.1.4 Detection of Unreachable Branches

Detecting unreachable branches is a vital task during SE, affecting the analysis's performance. While the proposed Algorithm 4 prevents the encoding of infeasible paths in Lines 8 to 26, the exploration strategy still tries to guide SE into paths that are not yet explored. If the paths are unreachable, they remain unexplored, and SE might not terminate if no other termination criteria, such as timeout or cycle bound, are defined.

Example 4.3: Unreachable Paths

Figure 4.7 shows a CFG with unreachable paths.

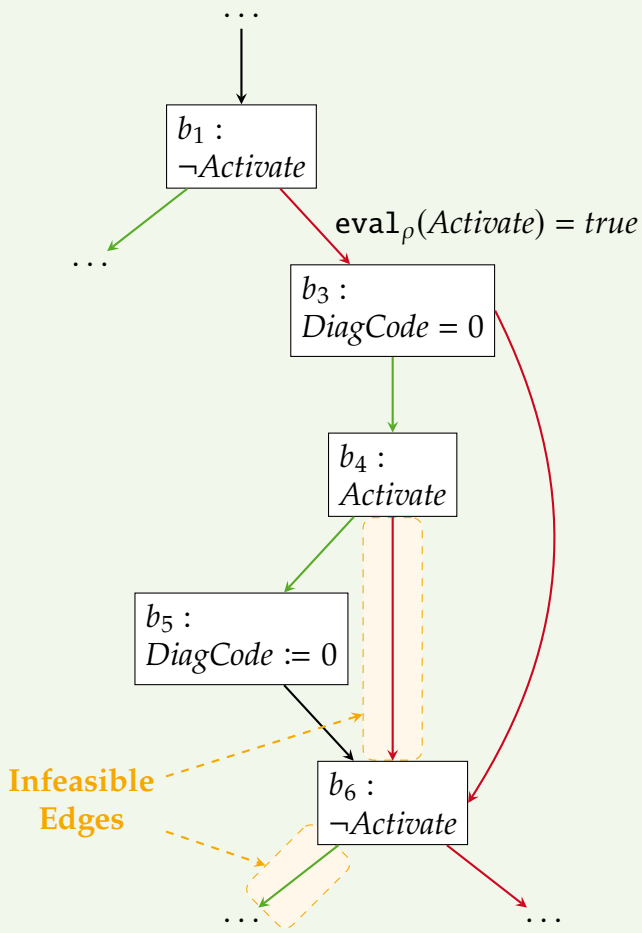


Figure 4.7: An excerpt of an implementation of the `SF_Antivalent` FB from the PLCopen Safety suite.

Once the execution reaches b_3, the value of the Boolean variable *Activate* is invariantly *true*. This renders the negative branch for the decision at b_4 to be infeasible for all execution contexts traversing b_3 and therefore makes the immediate path from b_4 to b_6 unreachable.

Currently, the SA of this thesis is not capable of performing abstract interpretation to detect unreachable branches. For this purpose, the algorithms from CRAB[1] are leveraged to build a value set analysis (VSA) as CRAB is a language-agnostic library for SA. VSA aims to compute the possible valuations a variable can take at each vertex in the CFG. This information can then be used to deduce statically whether a path is reachable or not.

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

In order to leverage Crab, the IR of this thesis needs to be transformed into the IR of Crab. While the static single assignment (SSA) pass has been abandoned for the use in SE (see Section 4.1.5), it is applied to the IR as part of the transformation pipeline to obtain a "Crab-analyzable" IR. The transformation pipeline consists of several steps. At first, the CFG is transformed using a basic block (BB) pass [Aho & Set+ 86]. Next, the instructions and expressions of the IR are transformed into three-address code (TAC) [Aho & Set+ 86]. Then, call instructions need to be rewritten to adhere to the expected internal structure of Crab's IR to be able to perform interprocedural SA. Last, the SSA pass is applied, and ϕ functions are resolved appropriately by pushing them into the predecessor vertices. Even though the information obtained by the ϕ functions is not exploited, the SSA pass is essential in assuring the unique variable versioning throughout all instructions and expressions. Finally, the transformed IR is marshaled into Crab's IR and the VSA is performed.

A precise numerical abstract domain has been chosen to derive sound bounds for the valuation of variables. While precise, the Boxes domain [Gur & Cha 10] is an expensive domain. The Boxes domain is sensitive to the number of "splits" with regard to each variable, which are introduced by Boolean operations and join operations at points where the control flow merges [Gro & Völ+ 22b]. Unfortunately, PLC programs typically exhibit a "state-machine" like behavior leading to a lot of "splits" on variables, and their cyclic dependency severely inhibits the application of Crab's VSA [Gro & Völ+ 22b]. Another downside in applying the transformation pipeline is the bloat-up of the CFG representation as the number of variables and vertices to consider significantly increases for the VSA.

To still reuse at least some information from the VSA, the analysis behavior can be tuned to achieve a trade-off between precision and run time. The Boxes domain was configured with a low widening delay and the decision to convexify the information after a certain amount of disjunctions resulting from splits. Overall, this made the analysis applicable, yielding usable but imprecise results.

4.1.5 Static Single Assignment and Variable Versioning

Typically, the IR is represented in SSA form [Bra & Buc+ 13]. While the transformation of the IR presented in this thesis to an equivalent SSA form has been efficiently implemented using a state-of-the-art algorithm [Bra & Buc+ 13], the introduction of ϕ functions has hindered the use of other algorithms. A ϕ function is placed at join points of the CFG and denotes the preceding definitions of a variable v reaching that specific vertex. For using an external SA tool, e.g., such as Crab, the ϕ functions had to be "pushed" back into the respective immediate predecessors. Hence, the SSA transformation pass was abandoned prior to performing SE. Instead, variables are versioned on the fly during SE. Consider the assignment $v := v + 1$ which is versioned using numerical subscripts as $v_{i+1} := v_i + 1$. This benefits merging and summary reuse as it is possible to distinguish between multiple

versions of the same variable and identify their modifications [Lin 17].

4.2 Generation of Summaries

Figure 4.8 gives an overview of the interplay between SE, summary generation, and the reuse of summaries generated from program version P in the reconfigured program version P'. This chapter focuses first on the SE implemented in this thesis

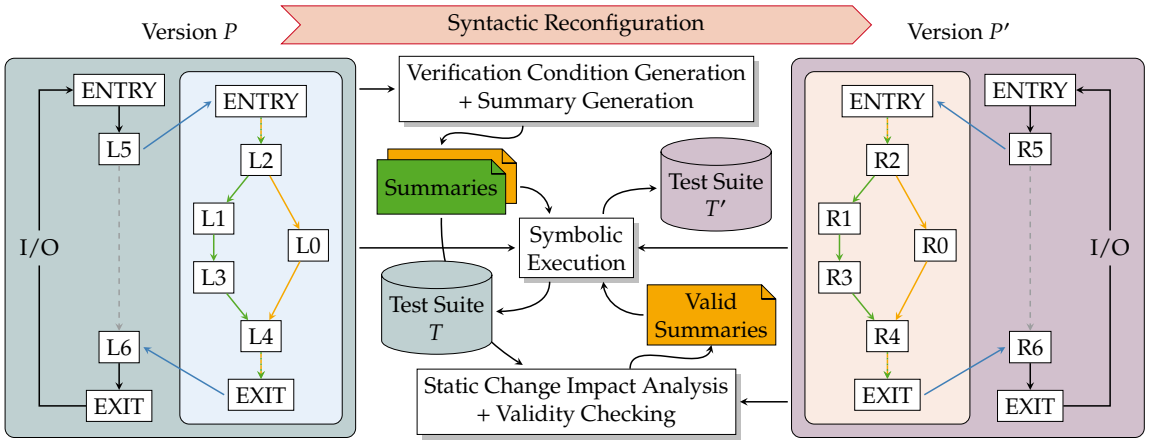


Figure 4.8: Overview of the interplay between symbolic execution, summary generation, and reuse across program versions.
Figure taken from Fig. 3 in [Gro & Völ+ 22a].

which is used to generate test cases adhering to branch coverage. Next, the VCG for generating summaries and the embedding in the SE is explained. Last, the reuse of the generated summaries and how to check that summaries are still valid after a program reconfiguration is illustrated.

The generation of summaries is detached from the SE [Lin & Mil+ 15; Lin & Mil+ 16]. The *summarization pass* identifies all code fragments suitable for summarization [Lin & Mil+ 16]. In order to summarize suitable code fragments, the algorithm of [Boh & Ham+ 18] is used to derive static verification conditions (VCs) for PLC programs. Currently, summarization is limited to whole FBs.

An alternative definition to the Definition 2.11 [God 07; God & Lah+ 11] in Section 2.4.4 is used to represent summarized paths and is presented next.

Definition 4.4: Summary Representation during VCG

A *summary* is a three-tuple $S = (\mathcal{B}, \mathcal{A}, \mathcal{H})$, where

- $\mathcal{B}$ represents the set of assumption literals encoding the flow of control through the summarized code fragment,
- $\mathcal{A}$ is the set of assumptions occurring on the summarized path,

- $\mathcal{H}$ is the set of hard constraints occurring on the summarized path.

Each summary uniquely characterizes its summarized path via its assumption literals $\mathcal{B}$ and the occurring assumptions $\mathcal{A}$ and hard constraints $\mathcal{H}$ on that path [Gro & Völ+ 22a]. This alternative representation was chosen as Definition 2.11 is not immediately applicable to CSE for PLC programs. The "statefulness" of FBs (see Section 2.1.1) conflicts with the loss of context information during summarization. As local and output variables may occur on the path constraint, there is no clear separation between inputs and state variables and, therefore, no clear separation between input pre- and output post-conditions. Instead of lowering and substituting valuations to adhere to Definition 2.11, the assumptions must also keep track of state variables as their valuations persist through multiple calls along the cycle [Gro & Völ+ 22a].

Different from Algorithm 2, the execution contexts are not merged when the control flow is joined. Instead, the CFG of the respective FB is explored depth-first, starting from the initial execution context and for each execution context forked during the execution until the exit vertex is reached. Each execution context reaching the exit of the FB represents a feasible execution path through that FB. Thus, summarization is done per path and illustrated in Example 4.4.

Example 4.4: Function Block Summarization

Figure 4.2a shows the CFG of the running example. The corresponding summarized paths are reflected by the respective color-coded summaries below.

$$\mathcal{B}_1 = \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_5 \rightarrow b_4, b_7 \rightarrow b_5, b_8 \rightarrow b_7, b_9 \rightarrow b_8, b_{11} \rightarrow b_9\}$$
$$\mathcal{A}_1 = \{b_5 \rightarrow (x_0 \geq 32), b_8 \rightarrow (y_1 \geq 3)\}$$
$$\mathcal{H}_1 = \{b_5 \rightarrow (y_1 = y_0 + 1), b_8 \rightarrow (z_1 = \mathit{true}), b_9 \rightarrow (y_2 = 0)\}$$

$$\mathcal{B}_2 = \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_5 \rightarrow b_4, b_7 \rightarrow b_5, b_{10} \rightarrow b_7, b_{11} \rightarrow b_{10}\}$$
$$\mathcal{A}_2 = \{b_5 \rightarrow (x_0 \geq 32), b_{10} \rightarrow \neg(y_1 \geq 3)\}$$
$$\mathcal{H}_2 = \{b_5 \rightarrow (y_1 = y_0 + 1), b_{10} \rightarrow (z_2 = \mathit{false})\}$$

$$\mathcal{B}_3 = \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_6 \rightarrow b_4, b_7 \rightarrow b_6, b_8 \rightarrow b_7, b_9 \rightarrow b_8, b_{11} \rightarrow b_9\}$$
$$\mathcal{A}_3 = \{b_6 \rightarrow \neg(x_0 \geq 32), b_8 \rightarrow (y_3 \geq 3)\}$$
$$\mathcal{H}_3 = \{b_6 \rightarrow (y_3 = y_0), b_8 \rightarrow (z_3 = \mathit{true}), b_9 \rightarrow (y_4 = 0)\}$$

$$\mathcal{B}_4 = \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_6 \rightarrow b_4, b_7 \rightarrow b_6, b_{10} \rightarrow b_7, b_{11} \rightarrow b_{10}\}$$
$$\mathcal{A}_4 = \{b_6 \rightarrow \neg(x_0 \geq 32), b_{10} \rightarrow \neg(y_3 \geq 3)\}$$
$$\mathcal{H}_4 = \{b_6 \rightarrow (y_3 = y_0), b_{10} \rightarrow (z_4 = \mathit{false})\}$$

The entry of a summarized FB is unconditionally part of every path through the FB and is denoted by the implication $b_3 \rightarrow \mathit{true}$. The conditions at branch points such as at b_4 and b_7 are pushed into the succeeding blocks, e.g., $b_5 \rightarrow (x_0 \geq 32)$, in correspondence with the encoding of the "`assume`" instruction in [Boh 21]. This is achieved by rewriting the branching instructions of the IR used in this thesis through pushing the respective expressions into the succeeding basic blocks.

Nonetheless, while the summary representation of Definition 4.4 facilitates the generation through the use of the assumption literals $\mathcal{B}$, it is of no use when trying to apply summaries during SE [Gro & Völ+ 22a]. The major problem lies in how paths are characterized, which is not invariant to potential reconfigurations of the PLC program, such as adding code. Therefore, the assumption literals are stripped from the summary representation, resulting, for example, in the following summary, which summarizes the right-outermost path in Figure 4.2a.

$$\mathcal{A}_4 = \{\neg(x_0 \geq 32) \wedge \neg(y_1 \geq 3)\}$$
$$\mathcal{H}_4 = \{(y_1 = y_0) \wedge (z_1 = \mathit{false})\}$$

To be reusable, the variable occurring in the assumptions and hard constraints are re-versioned while keeping the order of execution intact [Gro & Völ+ 22a]. In a wider sense, the assumptions denoted by $\mathcal{A}$ and the hard constraints denoted by $\mathcal{H}$ can be understood as "pre-" and "post-conditions" as defined in Definition 2.11, respectively. As this summary represents the equivalence class of paths that are rooted in concrete and feasible executions throughout the summarized FB, they are still considered to be under-approximating "must"-summaries [Gro & Völ+ 22a].

The reason for generating summaries before execution becomes apparent when looking at the application of summaries. Suppose summarization would occur during SE of Figure 4.2a. Assuming the 0-default initialization on the calling execution context resulting in the concrete store $\rho := \{x_0 \mapsto 0, y_0 \mapsto 0, z_0 \mapsto \mathit{false}\}$, the true branch succeeding b_7 may never be executed because there would always exist an applicable summary that takes the right-most outer path through the FB. To prevent this, the execution context can be executed to uncover further feasible paths along the CFG of the analyzed FB. However, additional execution, even though a summary is applicable, would circumvent the use of summaries in the first place. Nevertheless, in case of incomplete summarization, it is necessary as it is generally not statically deducible which path is taken before the actual execution.

4.3 Application of Summaries

Whenever a call is encountered, it is checked in Lines 27 to 35 of Algorithm 4 whether an applicable summary under the current execution context q exists or not. A summary is applicable if the assumptions and hard constraints of the summary conjoined with the current context's symbolic valuations reaching the summary's application point are satisfiable [Gro & Völ+ 22a]. In case multiple summaries are deemed applicable, an appropriate amount of execution contexts corresponding to the number of applicable summaries must be forked. The applicability check boils down to a query to a SMT solver.

Algorithm 5 findApplicableSummary

Input : Execution context $q \coloneqq (c, s = (b_\ell, \rho, \sigma, \pi), C)$, Set of summaries S
Local : SMT solver $\mathcal{S}$
Output : Set of applicable summaries $S_{applicable}$

1: $f \leftarrow q.\texttt{getFrame}()$
2: $G \coloneqq (\boldsymbol{V}, \ldots) \leftarrow f.\texttt{getCFG}()$
3: **for each** $v \in \boldsymbol{V}$ **do**
4: $\quad v_{reversioned} \leftarrow \texttt{reversion}(v)$
5: $\quad$ **for each** $v' \in \boldsymbol{V}$ **do**
6: $\quad\quad \gamma \leftarrow \sigma(v')$
7: $\quad\quad \gamma_{reversioned} \leftarrow \gamma.\texttt{substitute}(v, v_{reversioned})$
8: $\quad\quad \sigma_{reversioned}.\texttt{insert}\big(\langle v_{reversioned}, \gamma_{reversioned}\rangle\big)$
9: $\quad$ **end**
10: $\quad$ **for each** $\pi_i \in \pi$ **do**
11: $\quad\quad \pi_{reversioned}.\texttt{push}(\pi_i.\texttt{substitute}(v, v_{reversioned}))$
12: $\quad$ **end**
13: **end**
14: **for each** $s \in S$ of G **do**
15: $\quad \mathcal{S}.\texttt{add}(\sigma_{reversioned}, \mathcal{H}_s)$
16: $\quad \mathcal{S}.\texttt{add}(\pi_{reversioned}, \mathcal{A}_s)$
17: **end**

Checking whether a summary is applicable can be subdivided into two steps. First, the execution context q must be aligned with the summaries. Because summarization is a local analysis with regards to the summarized FB, it is given that no variables outside of the summarization's scope appear in the assumptions and hard constraints [Gro & Völ+ 22a]. The "re-versioning" thus is only limited to the interface of the callee's CFG, which is retrieved from the call stack C in Lines 1 to 2 of Algorithm 5. Therefore, the variables occurring in the symbolic valuations and the path constraint must be re-versioned to be aligned with the summaries' versions. This version alignment is necessary, as summaries are stored as reusable constraints without context, i.e., they do not adhere to any control flow other than the induced execution order due to the variable versioning.

Therefore, in Lines 3 to 9 of Algorithm 5, the symbolic valuations from the symbolic store σ are extracted, and each occurrence of the respective variable v is substituted by its "re-versioned" version $v_{reversioned}$. The re-versioning process aligns the identifiers of the flattened variables relative to version 0. For example, the symbolic valuations of the symbolic store $\sigma := \{x_7 \mapsto y_2 + z_5, x_9 \mapsto x_7 + 1\}$ of the execution context q reaching the summary's application point are reversioned to $\sigma_{reversioned} := \{x_0 \mapsto y_0 + z_0, x_1 \mapsto x_0 + 1\}$. As variables are versioned on the fly during SE, and their global versioning simulates a kind of SSA form, locally re-versioning to check for applicability does not interfere with the consistency of the global version store.

However, suppose the summary is indeed applicable. In that case, the variables must be appropriately re-versioned in Line 32 of Algorithm 4 to ensure consistency among other execution contexts q' waiting in the priority queue Q. The substitution of variables by their re-versioned version is also applied to the expressions occurring in the path constraint π. Next, the execution context's re-versioned symbolic valuations and the summary's hard constraints are added to the solver's assertion stack.

Example 4.5: Finding an applicable Summary

Consider the execution context $q = (c, s, C)$ reaching a summarization point, where $c = 2$, the call stack consists of only the main frame, and the path constraint and stores of the execution state look as follows:

$$\pi := \mathit{true}$$
$$\rho := \{x_0^2 \mapsto 0, \ldots\}$$
$$\sigma := \{x_0^2 \mapsto x_0^2, y_0^2 \mapsto \mathbf{ite}\big((x_0^1 \geq 32) \wedge \neg(y_4^1 \geq 3), y_4^1, y_2^1\big), \ldots\} .$$

Note that the upper index i of a variable x_j^i denotes the respective cycle number, and the lower index j is the implicit "SSA"-version. The value of y_0^2 is dependent on choices in prior cycles and contains further nested **ite**-expressions.

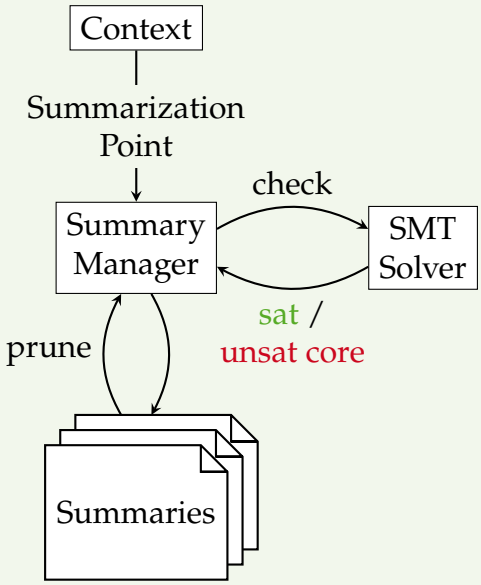


Figure 4.9: Overview of summary application.

The summary manager tries to find an applicable summary given the execution context by finding a satisfying assignment for the re-versioned versions of the summary's variables under the current path constraint and valuations of the current stores.
The summary manager evaluates each of the four previously generated summaries of Example 4.4, which are depicted in their re-versioned forms in Figure 4.10.

$$\mathcal{A}_1 = \{b_5 \rightarrow (x_0 \geq 32), b_8 \rightarrow (y_1 \geq 3)\}$$
$$\mathcal{H}_1 = \{b_5 \rightarrow (y_1 = y_0 + 1), b_8 \rightarrow (z_1 = \mathit{true}), b_9 \rightarrow (y_2 = 0)\}$$

$$\mathcal{A}_2 = \{b_5 \rightarrow (x_0 \geq 32), b_{10} \rightarrow \neg(y_1 \geq 3)\}$$
$$\mathcal{H}_2 = \{b_5 \rightarrow (y_1 = y_0 + 1), b_{10} \rightarrow (z_1 = \mathit{false})\}$$

$$\mathcal{A}_3 = \{b_6 \rightarrow \neg(x_0 \geq 32), b_8 \rightarrow (y_1 \geq 3)\}$$
$$\mathcal{H}_3 = \{b_6 \rightarrow (y_1 = y_0), b_8 \rightarrow (z_1 = \mathit{true}), b_9 \rightarrow (y_2 = 0)\}$$

$$\mathcal{A}_4 = \{b_6 \rightarrow \neg(x_0 \geq 32), b_{10} \rightarrow \neg(y_1 \geq 3)\}$$
$$\mathcal{H}_4 = \{b_6 \rightarrow (y_1 = y_0), b_{10} \rightarrow (z_1 = \mathit{false})\}$$

Figure 4.10: Re-versioned summarized paths of the FB depicted in Figure 4.2a.

The evaluation of the first summary yields a satisfying assignment, i.e., a valuation for the input variable $x_0 \mapsto 32$ that satisfies the assumptions and hard constraints of the summary while respecting the current context's path constraint and valuations. On the contrary, checking the third summary for applicability yields *UNSAT*, as the valuation of y_0^2 depends on the control flow choices of the prior cycles and under the 0-default initialization $y_0^0 \mapsto 0$ it is required that $x_0^2 \geq 32$ which contradicts the other clause of the summary's assumption.

An essential property of summaries is that the assumptions are the only reason an SMT solver may produce an unsatisfiable result. This means that for any valid execution context q with path constraint π, only the summary's assumptions can invalidate the valid execution context [Lin & Mil+ 16]. Formally, the constraints of a summary can be expressed and rewritten to [Lin & Mil+ 16]:

$$\begin{aligned}\phi &:= \underbrace{h_1 \wedge \ldots \wedge h_m}_{\text{hard constraints}} \wedge \underbrace{a_1 \wedge \ldots \wedge a_n}_{\textit{assumptions}} \\ &= h_1 \wedge \ldots \wedge h_m \wedge (a_1 \vee b_1) \wedge \ldots \wedge (a_n \vee b_n) \wedge \neg b_1 \wedge \ldots \neg b_n \ ,\end{aligned}$$

where $b_1, \ldots, b_n$ denote n free Boolean assumption literals. This implies that hard constraints can never invalidate a valid execution context. The only reason for a valid execution context to be invalid is if a conflicting assumption is added, resulting in a solver check failure [Lin & Mil+ 16]. Furthermore, this way of representing summaries gives rise to an efficient form of representation presented in Example 4.6.

Example 4.6: Representation of Summaries as a Trie

Figure 4.11 illustrates the computed summaries of Example 4.4 as a trie [Fre 60]. Hard constraints are depicted as rectangles and assumptions as rectangles with rounded corners. A trie, derived from the word retrieval, is an efficient tree data structure for searching within prefix-based data [Fre 60]. The weakest-precondition "*true*" approximates the root node of the summary without compromising the correctness of the summary [God 07].

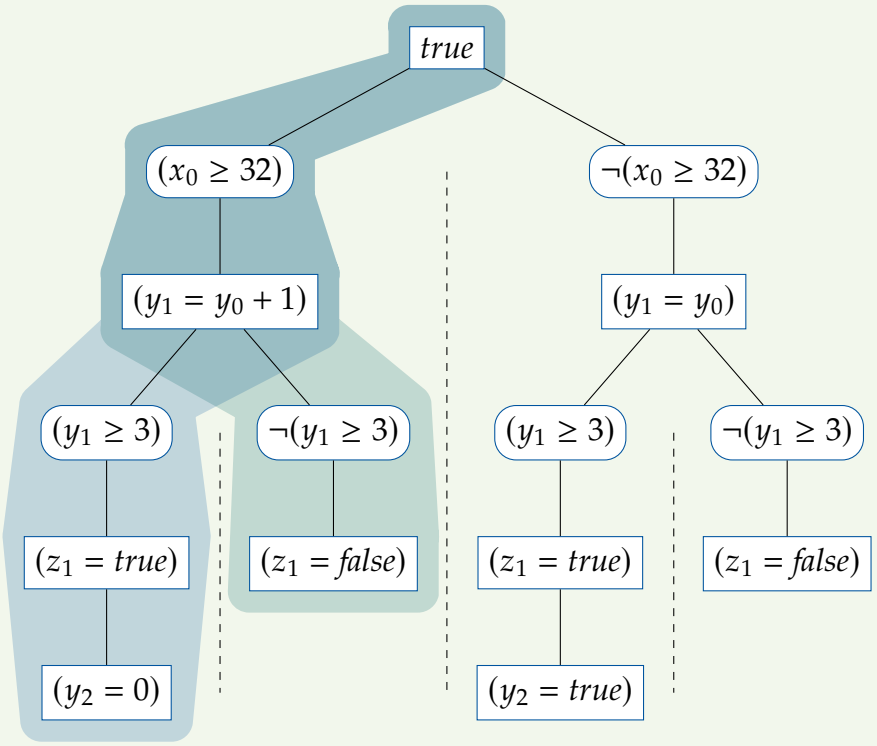


Figure 4.11: Exemplary summary depicted as a trie.

The representation of summaries as a trie has several benefits. First and foremost, it is an efficient way to store shared clauses in memory as highlighted in Figure 4.11 by the blue and petrol-colored subtries representing different execution paths throughout the CFG in Figure 4.2a; second, it gives rise to incremental checking.

Through the use of the incremental solving interface of Z3, an unsatisfiable core can be obtained in case the summary is not applicable under the current execution context. An *unsatisfiable core* denotes a subset of assertions that prevent the derivation of the satisfiability of a formula. The unsatisfiable core returned by the solver can be used to prune the summaries by excluding the summaries that share the same assumptions to speed up the applicability checking. Concerning the trie depicted in Example 4.6, the corresponding subtrees can be "pruned" away as summarization is an analysis local to the respective FB [Gro & Völ+ 22a]. While the removal of the corresponding assumption literals in the trie representation aggravates summary pruning, it is necessary when trying to determine whether a summary is reusable or not. As code might be added or removed, the correspondence between the assumption literals and the structure of the CFG is not kept, and this mismatch leads to inconclusive results when applying a summary [Gro &

Völ+ 22a].

Although summary pruning is theoretically possible when, e.g., following the execution model of SMART [God 07], as each run is along a single path and concrete valuations are available, it can not be applied to the algorithm proposed in this thesis. This is a trade-off to the current merge strategy as there potentially do not exist viable concrete valuations for the summary's inputs due to prior merging. Therefore, summary pruning is only feasibly usable if concrete valuations for the respective variables of all paths are present in the current execution context. However, experiments have shown (cf. Chapter 6) that merging is, in general, a suitable heuristic to limit the path explosion due to the cyclic nature of PLC programs.

If the solver returns "sat", the summary is applicable under the current execution context q [Gro & Völ+ 22a]. After determining which summaries are applicable and which are not, the current execution context q must be updated to account for the effect of the summary. For each applicable summary $s \in S_{applicable}$, a new execution context q' is derived in Lines 31 to 33 of Algorithm 4. Summary application proceeds by first re-versioning the hard constraints and assumptions of the summary to align them with the local and global execution context similar to Lines 3 to 9 of Algorithm 5. Then, to reflect the effect of the summary and fast-forward execution to the exit vertex of the summarized FB [Gro & Völ+ 22a], the assumptions are added to the path constraint π. The valuations from the hard constraints are used to update the symbolic store of the execution context.

4.4 Reusing Summaries across Program Versions

Once a PLC program has been reconfigured as visualized in Figure 4.2, the aim is to reuse as much work as possible from the previous version. For this purpose, summaries generated during the SE of the previous version should be checked for validity in the reconfigured version. This problem is defined as the must-summary checking problem.

Definition 4.5: Must Summary Checking Problem [God & Lah+ 11]

Given a program P, a set of summaries S for that program, and its reconfigured version P'. Which summaries $s \in S$ are still valid must summaries for the reconfigured program P'?

In the following, a three-phased algorithm [God & Lah+ 11] which statically analyzes whether a summary is applicable or not, is presented. Figure 4.12 illustrates the three-phased algorithm, which is either executable in isolation or can be run in a pipeline. The input to this pipeline are all summaries of the prior program version, and the output is a set of safely reusable summaries which are still valid in the new program version. Each phase is explained in the subsequent sections with the help of Algorithm 6.

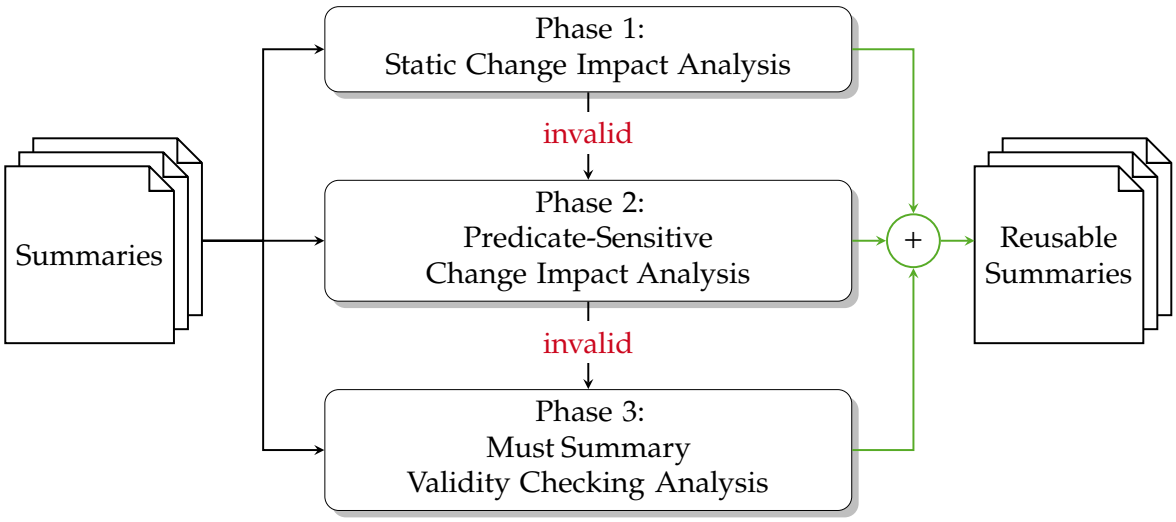


Figure 4.12: Three-phased summary reuse checking algorithm [God & Lah+ 11]. *Figure adapted from Fig. 5 in* [Gro & Völ+ 22a].

Algorithm 6 Statically Validating Must Summaries [God & Lah+ 11]

Input : CFG G, Summaries S
Output : Validated summaries S_{valid}

```
1:  for each callee G' ∈ callGraph(G) do
2:      // Phase 1
3:      B_change-annotated ← getChangeAnnotatedVertices(G')
4:      if B_change-annotated = ∅ then
5:          S_valid ← S
6:      else
7:          vcg_old ← generateVerificationConditions(G'_old)
8:          vcg_new ← generateVerificationConditions(G'_new)
9:          for each s ∈ S do
10:             // Phase 2
11:             valid ← predicateSensitiveCIA(s, vcg_old, B_change-annotated)
12:             if valid then
13:                 S_valid.push(s)
14:             else
15:                 // Phase 3
16:                 valid ← validityChecking(s, vcg_new)
17:                 if valid then
18:                     S_valid.push(s)
19:                 end
20:             end
21:         end
```

22: | **end**
23: **end**
24: **return** S_{valid}

4.4.1 Static Change Impact Analysis

The first phase performs a static CIA on each callee in the call graph (CG) of the program P to find reconfigurations and their implications on the other CFGs. An instruction in the CFG of P is reconfigured if it is either changed or deleted in the subsequent version P' or the ordered set of immediate successors has changed [God & Lah+ 11]. This translates directly to the CAP representation of this thesis. Thus, a CFG is reconfigured if it contains a reconfigured instruction or calls a reconfigured CFG. Example 4.7 illustrates the impact of reconfigurations in the CG.

Example 4.7: Call Graph and Impact of Reconfigurations

Figure 4.13 shows an exemplary CG and the impact of reconfigurations on the validity of other FBs' summaries.

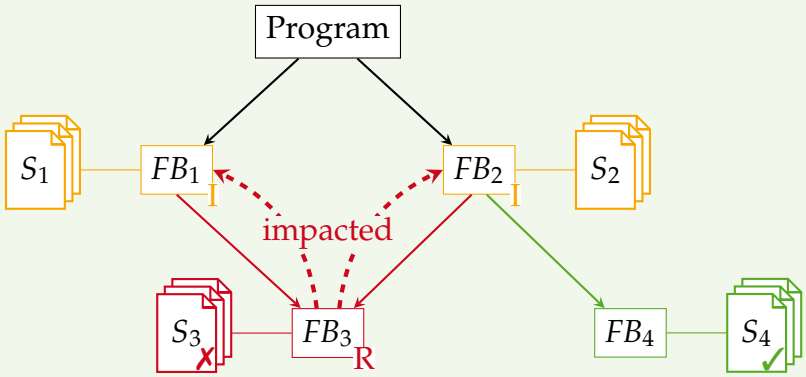


Figure 4.13: Call graph with reconfigured CFGs and the implications for summaries.

In this example, the CFG of FB_3 contains one or more reconfigured instructions and hence is flagged with R for "reconfigured". By traversing the CG through the dependencies on FB_3, the impacted FBs, FB_1 and FB_2, can be determined and also flagged as I for "impacted" by the reconfiguration. This is because their summaries use the summaries from the reconfigured CFG. Solely the summaries of FB_4 are deemed valid, as the CFG was not reconfigured in any way.
Please note that the contribution of this thesis only considers call stacks of depth 1, which is reflected in Lines 3 to 5 of Algorithm 6 and is only a limitation of the current underlying implementation and not the algorithm of [God & Lah+ 11] in general. Therefore, no nested summaries can occur.

Reconfigurations of the CFG are represented by `change`-annotation macros in the IR. An FB in P' is therefore reconfigured if it contains such a `change`-annotated expression. Line 3 of Algorithm 6 traverses the structure of the CFG and retrieves all vertices whose instructions include a `change`-annotated expression. The subsequent Line 4 checks whether this set is empty or not. In case it is empty, the CFG does not contain `change`-annotated vertices, and all summaries of the prior program version P are also valid in the subsequent program version P'. They can be safely reused, and the valid summaries are collected in Line 5. If the set of `change`-annotated vertices is not empty, all the summaries for this specific CFG are invalidated by the first phase.

This is an imprecise but inexpensive over-approximation which implies that the syntactic reconfigurations also have a potential semantic effect [Gro & Völ+ 22a]. The algorithm then checks the next callee in the CG until the whole CG has been analyzed.

4.4.2 Predicate-Sensitive Change Impact Analysis

Even though the CFG is reconfigured, the reconfiguration might not introduce semantic effects that lead to an invalidation of a summary. Therefore, the second phase analyzes the "pre-" and "post-conditions" of the summary and is thus predicate-sensitive CIA [God & Lah+ 11]. Instead of over-approximately looking at all paths between the entry and exit of the summary, only the paths represented by the assumptions and hard constraints are taken into account [Gro & Völ+ 22a]. The predicate-sensitive analysis is performed on the static VCs of the old program P to check whether a reconfiguration exists on these specific paths.

Generation of Verification Conditions

The generation of VCs in Lines 7 to 8 of Algorithm 6 resembles Algorithm 1. It also symbolically encodes the PLC program's semantics while using a variant of the Cycle-BMC procedure presented in [Boh & Ham+ 18; Boh 21]. Instead of encoding a whole cycle, the procedure encodes the passed CFG from the entry up to its exit vertex. In fact, it reuses the algorithm used for the summary generation in Section 4.2. By passing an appropriate argument to the encoding algorithm, the new or old version of the CFG is encoded. Exemplary results of the VCG for the running example's old program version are shown in Example 4.8.

Example 4.8: Verification Conditions for the Running Example

The following VCs result from the encoding of the old program version's CFG illustrated in Figure 4.2b.

$$
\begin{aligned}
\mathcal{B} :=\ & \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_5 \rightarrow b_4, b_6 \rightarrow b_4 \\
& b_7 \rightarrow b_5 \vee b_6, b_8 \rightarrow b_7, b_9 \rightarrow b_8, b_{10} \rightarrow b_7, \\
& b_{11} \rightarrow b_9 \vee b_{10}\} \\
\mathcal{A} :=\ & \{b_5 \rightarrow (x_0 \geq 32), b_6 \rightarrow \neg(x_0 \geq 32), \\
& b_8 \rightarrow (y_3 \geq 3), b_{10} \rightarrow \neg(y_3 \geq 3)\} \\
\mathcal{H} :=\ & \{b_5 \rightarrow y_1 = y_0 + 1 \wedge y_3 = y_1, b_6 \rightarrow y_2 = y_0 \wedge y_3 = y_2, \\
& b_8 \rightarrow z_1 = \mathit{true}, b_9 \rightarrow y_4 = 0 \wedge y_5 = y_4 \wedge z_3 = z_1, \\
& b_{10} \rightarrow z_2 = \mathit{false} \wedge y_5 = y_3 \wedge z_3 = z_2\}
\end{aligned}
$$

Note that auxiliary valuations such as $y_3 = y_2$ and $y_5 = y_3$ are the result of merging. They ensure that the correct valuations are propagated depending on the control flow.

After the VCs are generated, they are instrumented to check for validity. The instrumentation partially follows the instrumentation presented in [God & Lah+ 11]. First, an auxiliary Boolean variable *reconfigured* is added. This auxiliary Boolean variable is used to denote vertices in the old program version P, which differ from the new program version P'. For each `change`-annotated vertex $b_\ell \in B_{\textit{change-annotated}}$, a hard constraint $b_\ell \rightarrow (\textit{reconfigured} = \textit{true})$ implying the auxiliary Boolean variable *reconfigured* is added. In case such a `change`-expression occurs in a conditional instruction, the corresponding assignment is added before that instruction to the respective basic block [Gro & Völ+ 22a]. The hard constraints of the entry of the VCs are augmented with $b_{\ell_e} \rightarrow (\textit{reconfigured} = \textit{false})$ to account for the change of the auxiliary variable. In addition, two auxiliary assumption literals, A and H are introduced, representing the assumptions and hard constraints of the summary under analysis. The goal is to check whether the instrumented static VCs have a satisfiable solution under those auxiliary assumption literals. If these assertions hold, then the summary is a valid must summary for all possible inputs of the old program and is added to the set of valid summaries $S_{\textit{valid}}$ in Line 13 of Algorithm 6. It can safely be reused in the new program version P' as there are no reconfigured instructions on the path characterized by this specific summary [Gro & Völ+ 22a]. In Example 4.9 a summary is exemplarily analyzed for validity using the proposed instrumentation of the VCs for the predicate-sensitive analysis.

Example 4.9: Running Example and Phase 2

In order to correctly adhere to the flow of control, the assumptions $\mathcal{A}$ and hard constraints $\mathcal{H}$ of the summary under analysis must be lowered and

re-versioned, respectively. In this example, the summary S_2 of Example 4.4

$$\mathcal{A}_2 = \{b_5 \rightarrow (x_0 \geq 32), b_{10} \rightarrow \neg(y_1 \geq 3)\}$$
$$\mathcal{H}_2 = \{b_5 \rightarrow (y_1 = y_0 + 1), b_{10} \rightarrow (z_1 = \mathit{false})\}$$

is checked for validity using the predicate-sensitive analysis. The VCs of the current context in the program version before reconfiguration are given in Example 4.8. As the summary S_2 writes to the local variable y, the corresponding assumptions are lowered to reflect the effect of the hard constraints. Furthermore, the hard constraints of the summary are re-versioned to account for the effect of the VCs yielding the following instrumentation:

$$\begin{aligned}
b_3 &\rightarrow (\mathit{reconfigured}_0 = \mathit{false}) \\
b_5 &\rightarrow (\mathit{reconfigured}_1 = \mathit{true}) \wedge (\mathit{reconfigured}_2 = \mathit{reconfigured}_1) \\
b_6 &\rightarrow (\mathit{reconfigured}_2 = \mathit{reconfigured}_0) \\
A &\rightarrow (x_0 \geq 32) \wedge \neg(y_0 > 2) \\
H &\rightarrow (y_5 = y_0 + 1) \wedge (z_3 = \mathit{false}) \\
b_{11} \rightarrow H &\implies \neg(\mathit{reconfigured}_2)
\end{aligned}$$

Under the assumption that A and H are *true*, and the exit b_{11} is reachable, the SMT solver returns *UNSAT*. The reason is that once A is assumed, the only possible path at b_4 is through b_5, as the valuation of x_0 must satisfy $x_0 \geq 32$. This leads to a conflict, as *reconfigured* is set to *true*, but H enforces that the variable must be *false* at the exit of the FB. While the summary may still be semantically applicable, the second phase solely checks whether a reconfiguration occurs on the path and invalidates it in that case. Do note that the introduction of the auxiliary *reconfigured* variables for path distinction due to merging does not matter, as the analysis is over-approximating and a conflict refers to a path toggled by the assumptions A.

4.4.3 Must Summary Validity Checking Analysis

In the third phase, the summaries are checked for applicability using the generated VCs of the new program version P'. In contrast to the check in the second phase, in which reconfigurations occurring on paths were detected, this phase applies a semantic check. The generated VCs are checked under the summaries' assumptions and hard constraints by instrumenting the VCs of the new program version similarly to the second phase. While in [God & Lah+ 11], an additional auxiliary variable is introduced to track whether the reconfigured code fragment reaches the exit vertex of the summarized FB, it is disregarded in this thesis. This "simplification" is based on the assumption that all executions throughout an FB terminate. If the

VCs and the constraints from the summary have a satisfying solution, the summary is added to the set of valid summaries in Line 18. This is an absolute guarantee that the summary is reusable in the new version [God & Lah+ 11]. The derived VCs and the instrumentation for the running example in Figure 4.2b are shown in Example 4.10.

Example 4.10: Running Example and Phase 3

This example considers summary S_2 from Example 4.9. The VCs of the current execution context in the reconfigured program version are depicted below.

$$\begin{aligned}
\mathcal{B} := \; & \{b_3 \rightarrow \mathit{true}, b_4 \rightarrow b_3, b_5 \rightarrow b_4, b_6 \rightarrow b_4 \\
& b_7 \rightarrow b_5 \vee b_6, b_8 \rightarrow b_7, b_9 \rightarrow b_8, b_{10} \rightarrow b_7, \\
& b_{11} \rightarrow b_9 \vee b_{10}\} \\
\mathcal{A} := \; & \{b_5 \rightarrow (x_0 \geq 32), b_6 \rightarrow \neg(x_0 \geq 32), \\
& b_8 \rightarrow (y_3 \geq 3), b_{10} \rightarrow \neg(y_3 \geq 3)\} \\
\mathcal{H} := \; & \{b_5 \rightarrow (y_1 = y_0 + 2) \wedge (y_3 = y_1), b_6 \rightarrow (y_2 = y_0) \wedge (y_3 = y_2), \\
& b_8 \rightarrow (z_1 = \mathit{true}), b_9 \rightarrow (y_4 = 0) \wedge (y_5 = y_4) \wedge (z_3 = z_1), \\
& b_{10} \rightarrow (z_2 = \mathit{false}) \wedge (y_5 = y_3) \wedge (z_3 = z_2)\}
\end{aligned}$$

Due to the reconfiguration, the value of y is incremented by 2 at location b_5. The instrumentation boils down to the following additional assertions added to the solver's stack:

$$\begin{aligned}
A &\rightarrow (x_0 \geq 32) \wedge \neg(y_0 > 2) \\
H &\rightarrow (y_5 = y_0 + 1) \wedge (z_3 = \mathit{false})
\end{aligned}$$

As the assumptions of the summary of the old program version drive the execution through b_5 and b_{10} via the assertions $(x_0 \geq 32)$ and $\neg(y_1 \geq 3)$, the hard constraints at the exit of the FB are not consistent anymore with the new VCs. The conflicting clauses are `new`: $(y_5 = y_0 + 2)$ and `old`: $(y_5 = y_0 + 1)$ and thus *UNSAT* is returned by the SMT solver. This is a limitation of "must"-summaries as they are rooted in concrete executions.
As this summary has been invalidated by all three phases it is not reusable for application checking during SE of the new program version.

Regeneration of Invalidated Summaries

While all validated summaries S_{valid} from all three phases can be reused in the new program version P', all invalidated summaries are not reusable. Hence, there exist paths in the new program which are not characterized by a summary yet. To circumvent this problem appropriately, those paths must be "re-summarized".

With the same reasoning as generating the summaries before the execution, regeneration also has to occur before the execution of the new program version P'. In order to prevent redundant work, the generated VCs of the new program version obtained during the static validation of the summaries can be reused to regenerate those missing summaries. In its current state, regeneration can partially regenerate summaries using the same technique described in Section 4.2. However, it does not yet exploit the knowledge of which summaries are valid and invalid, and also unreachable branches complicate the regeneration and is thus not in the scope of this thesis.

Concluding Remarks This chapter has introduced and explained a compositional SE algorithm for automatic TSG. Several heuristics and design decisions have been presented and explained in detail, as the proposed CSE serves as a baseline for the techniques presented in the succeeding chapter on TSA. A heuristic for the improvement of subsequent analysis in the form of reusable FB summaries has been presented in the second part of this chapter. The proposed algorithms are evaluated in Chapter 6 in the respective section Section 6.2.

5 Test Suite Augmentation

This chapter presents a technique for augmentation of the test suite for a reconfigured program. Figure 5.1 gives an overview of the discussed contents in the

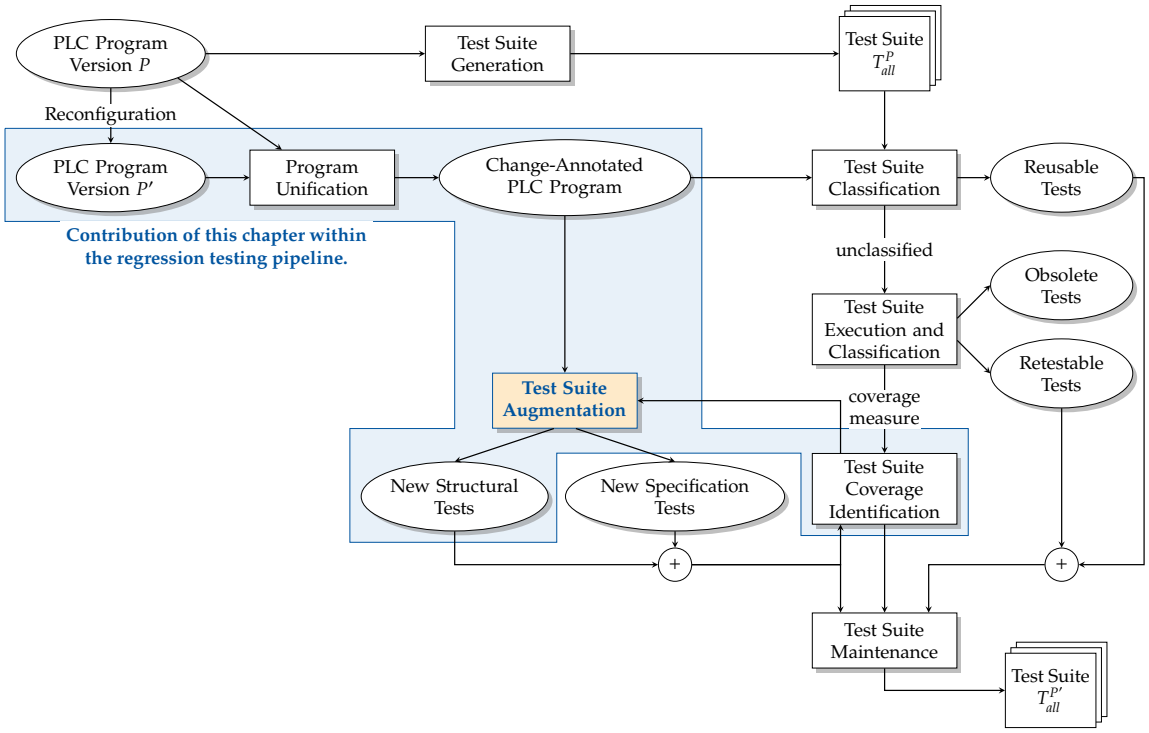


Figure 5.1: Overview of the regression testing pipeline and contribution of this chapter.
Figure adapted from Figure 3 in [Leu & Whi 89].

subsequent sections. In the beginning, the core theoretical foundation and the idea behind TSA are given by examining the test suite coverage identification problem. The test suite coverage identification problem is particularly interesting for determining whether the current test suite T is adequate enough to test the reconfigured program version P'. Next, SSE is presented and discussed in-depth, as it is the main algorithm that drives the TSA.

Motivation To motivate the need for TSA, an exemplary program and its reconfiguration is illustrated in Figure 5.2. The old test suite consists of valid inputs which

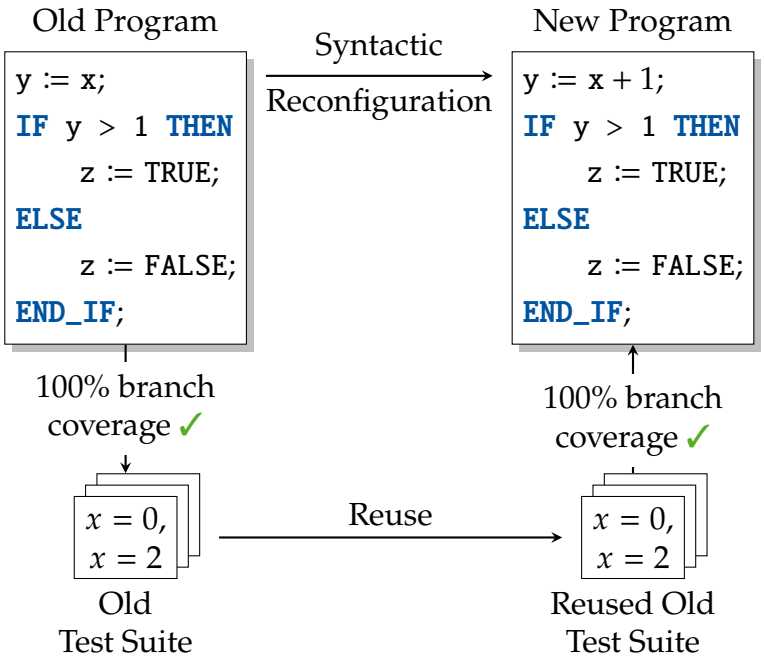


Figure 5.2: Motivating example.
Example adapted from Fig. 3 in [Kuc & Pal+ 18].

are homogeneous concerning the feasible program paths of the old program [Wey & Jen 91]. While executing the old test suite on the new program yields the same coverage, one has to keep in mind that coverage alone does not quantify the capability of the test suite to reveal regressions. Instead, one is interested in test suites homogeneous with regards to failure, where this thesis defines failure to occur if an input exposes a behavioral difference.

Definition 5.1: Behavioral Difference [Nol 20]

In general, behavioral difference refers to a difference in the execution behavior of a program. Such a difference has several forms, e.g., a difference in execution time, a difference in the traversed code fragments, or a difference in the outputs of a program.

The goal of the subsequent sections is to derive test cases that expose behavioral differences between the old and the reconfigured program.

Definition 5.2: Difference-Revealing Test Case

Given an old program version P and its reconfigured version P'. A test case is said to be difference-revealing if different output valuations are produced after executing both program versions with the same input valuations.

Hence, this chapter aims to derive test inputs that expose difference-revealing behavior in the outputs of the programs, as displayed in Figure 5.3.

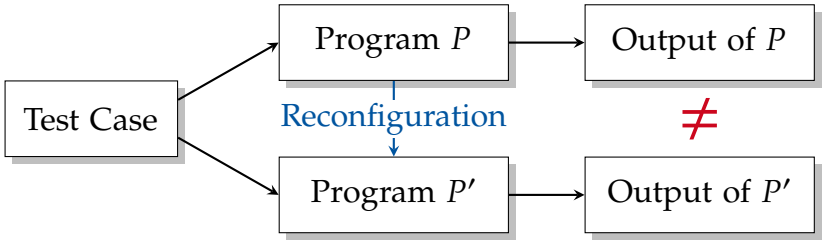


Figure 5.3: A test case producing difference-revealing outputs when executed on both program versions.

Running Example Throughout this chapter, the following running example will be used, and the following examples refer to this small but expressive PLC program. The TSG of Chapter 4 has generated the following test suite $T :=$

```
1  PROGRAM P'
2  VAR_INPUT
3      x : INT;
4  END_VAR
5  VAR
6      y : INT;
7  END_VAR
8  VAR_OUTPUT
9      z : BOOL := FALSE;
10 END_VAR
11     y := change(x, x + 1);
12     IF y > 1 THEN
13         z := TRUE;
14     ELSE
15         z := FALSE;
16     END_IF;
17 END_PROGRAM
```

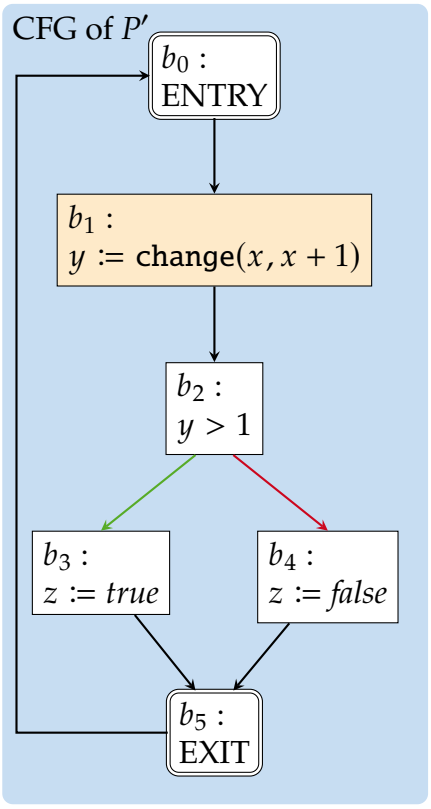


Figure 5.4: Textual and graphical representation of the running example.

$\{\underbrace{(x \mapsto 0, z \mapsto \mathit{false})}_{t_1}, \underbrace{(x \mapsto 2, z \mapsto \mathit{true})}_{t_2}\}$, which is used in the presented analyses.

5.1 Test Suite Coverage Identification Problem

An essential task in testing is to assess whether a set of components in the program are covered or not. For this purpose, a *test data adequacy criterion* $C = (\eta, \nu)$ is defined [Rot 96]. A typical code-based adequacy criterion is statement or branch coverage, where η specifies the set of vertices in the CFG and ν specifies how to execute the instruction associated with the vertices [Rot 96]. For instance, a branch-covering test suite T is adequately enough for a program P if, for all branching vertices (as specified by η), there exists at least one test case (as specified by ν) by which it is exercised [Rot 96]. Now, after reconfiguring the program, this test suite T might not be adequate anymore to cover the reconfigured program P'. In order to distinguish between components that need to be retested and components that do not need to be retested, a significance criterion is added to the definition of the test adequacy criterion leading to the following definition of the regression test adequacy criterion:

Definition 5.3: Regression Test Adequacy Criterion [Rot 96]

Given a program P and a reconfigured program P', a *regression test adequacy criterion* is a 3-tuple $C = (\eta, \nu, \alpha)$ such that

- η specifies a set of vertices of P',
- ν specifies what it means to execute a vertex in P',
- α specifies what it means for a vertex of P' to be significant.

The regression test adequacy criterion must be appropriately initialized to augment the test suite T with new structural tests. SSE is a technique that augments the test suite T by deriving difference-revealing test cases, where η specifies a set of change-annotated vertices of P', ν specifies how change-annotated vertices are reached, and α categorizes divergent execution contexts as significant. The problem of finding these significant execution contexts is defined in Definition 5.4.

Definition 5.4: Test Suite Coverage Identification Problem [Rot 96]

Given a program P and the reconfigured program P'. Let $C = (\eta, \nu, \alpha)$ be a regression test adequacy criterion and Q the set of components of P' specified by η. Find a maximal $Q' \subseteq Q$, such that $\forall q \in Q' : q$ is significant as specified by α.

The test coverage identification problem now involves finding the maximum number of test cases that lead to a divergence in the new program version P' with regard to the change-annotated vertices of η.

Because SSE is a technique that is capable of answering this coverage identification problem, it is explained throughout the subsequent sections.

5.2 Shadow Symbolic Execution

Before going in-depth and explaining the internal working of SSE, a high-level overview of how TSA is embedded in the development process is given.

5.2.1 Developer-centered Test Suite Augmentation Process

Figure 5.5 gives an overview of the developer-centered TSA process. The developer

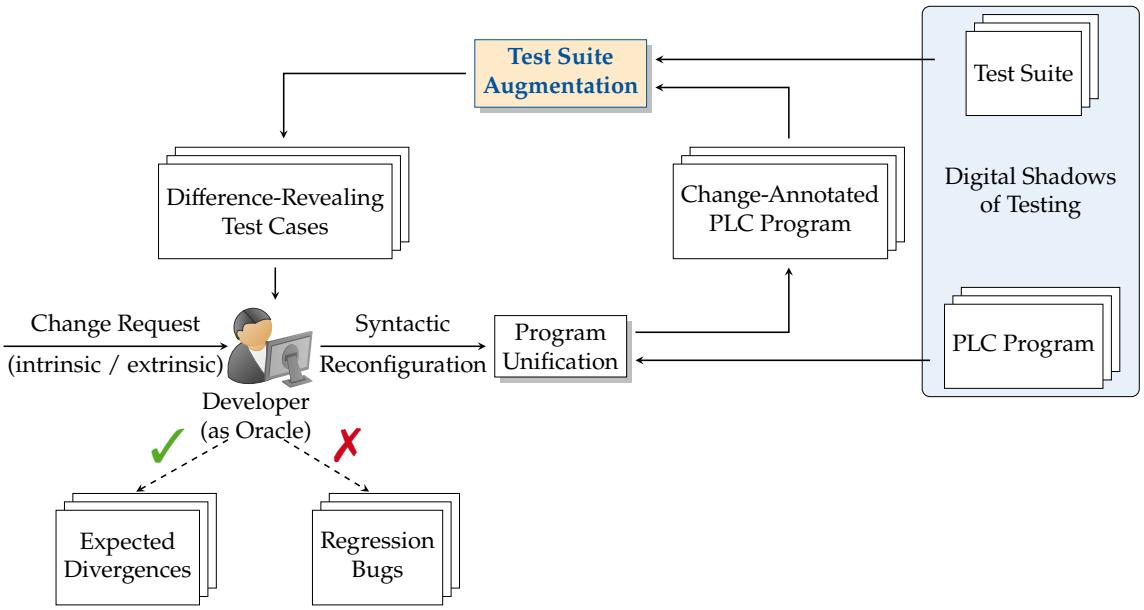


Figure 5.5: Overview of the developer-centered TSA process.
Figure adapted from Fig. 3 in [Gro & Völ+ 22b].

obtains either an intrinsic or extrinsic change request. Intrinsic change requests usually refer to software correction tasks (cf. Section 1.3). In contrast, extrinsic change requests stem from changing customer requirements or market demands leading to an enhancement of software (see Figure 1.10 in Section 1.3). The developer's task is to unify the old and new program versions into a CAP. Unification is done with the aid of the `change`-annotation macro presented in Section 2.5. 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 control- and data 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. TSA amounts to four phases which are applied subsequently and are depicted in Figure 5.6. The SSE algorithm is performed on the CAP and the test artifacts of the previous program version P. The results are used to perform a BSE afterward, potentially yielding divergent test inputs. These test inputs are replayed and simulated on the new program version P' in order to derive *difference-revealing* test cases as defined

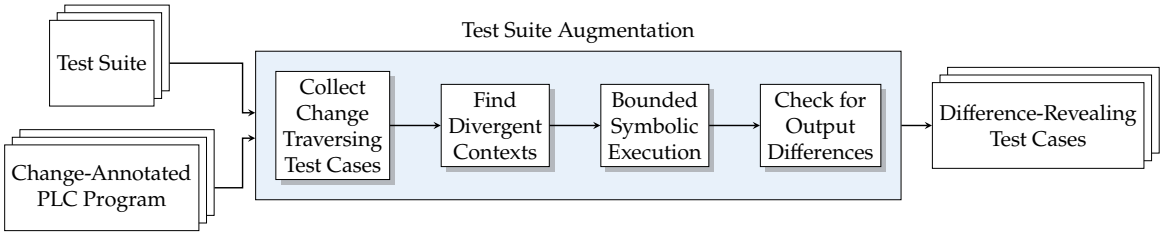


Figure 5.6: Overview of TSA using SSE.
Figure adapted from Fig. 4 in [Kuc & Pal+ 18].

in Definition 5.2. These test cases can be examined by a developer who serves as an oracle to categorize them as either expected divergences, e.g., because a bug was fixed, or actual regressions.

An interesting research question in this context is whether the four-way forking concept is applicable to the PLC domain. The introduced baseline BSE of Chapter 4 is augmented with the functionality of the SSE [Kuc & Pal+ 18] algorithm. In general, it can be intractable because outputs are potentially difference-revealing after k cycles (depending on the internal state of the PLC program). Therefore, the analysis may run out of memory before the divergence is reached. SSE is essentially just SE at its core with a little bit of extra, and that extra is the concept of four-way forking [Kuc & Pal+ 18].

The TSA algorithm pipeline of Figure 5.6, which uses SSE, is presented in Algorithm 7.

Algorithm 7 Test Suite Augmentation using SSE [Kuc & Pal+ 18]

Input : CAP $P = (G, \mathcal{G})$, CFG $G = (\boldsymbol{V}, \boldsymbol{V}_{input}, (B, E), b_{\ell_e}, b_{\ell_x})$, Test Suite T
Local : Divergent execution contexts and triggering test cases $Q_{divergent}$
Output : Difference revealing test cases $T_{difference\text{-}revealing}$

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

In order to account for "shadows", i.e. `change`-annotated expressions (see Example 5.2), occurring in the concrete and symbolic store of the execution state s, Definition 2.10 is extended in Definition 5.5.

Definition 5.5: Divergent Execution State

A *divergent execution state* is a tuple $s_{divergent} = (s, \rho_s, \sigma_s, \pi_{old}, \pi_{new})$, where

- $s = (b_\ell, \rho, \sigma, \pi)$ is an execution state,
- ρ_s is a *shadowed concrete store* that maps shadow variables to expressions over concrete values,
- σ_s is a *shadowed symbolic store* that maps shadow variables to expressions over concrete and symbolic values,
- π_{old} and π_{new} denoting the *path constraint* of the old and new program version, P and P', respectively, representing a set of conditional expressions taken on the execution path up to vertex b_ℓ.

In order to correctly update the state, the evaluation functions $\text{eval}_\rho(e)$ and $\text{eval}_\sigma(e)$ (see Section 2.3) are lifted to concrete and symbolic shadow expressions via $\text{eval}_\rho^{shadow}(e)$ and $\text{eval}_\sigma^{shadow}(e)$, respectively.

Definition 5.6: Shadow Evaluation Function

Given a concrete or symbolic shadow store, ρ_s and σ_s, respectively, the shadow evaluation functions $\text{eval}_\rho^{shadow} : \boldsymbol{E} \to \boldsymbol{D}$ and $\text{eval}_\sigma^{shadow} : \boldsymbol{E} \to \boldsymbol{\Sigma}$ are defined recursively in terms of their respective concrete and symbolic evaluation functions for non-shadowed variables occurring in the expression e. Depending on whether the *old*, *new*, or *both* versions of an expression containing a change-annotation should be evaluated, the respective shadowed stores ρ_s and σ_s are used.

Furthermore, the introduction of the divergent state gives rise to the definition of a divergent execution context.

Definition 5.7: Divergent Execution Context

A *divergent execution context* is a tuple $q_{divergent} = (c, s_{divergent}, C)$, where

- $c \in \mathbb{N}$ denotes the current execution cycle of the PLC program,
- $s_{divergent} = (s, \rho_s, \sigma_s, \pi_{old}, \pi_{new})$ is a divergent execution state,
- C is a call stack.

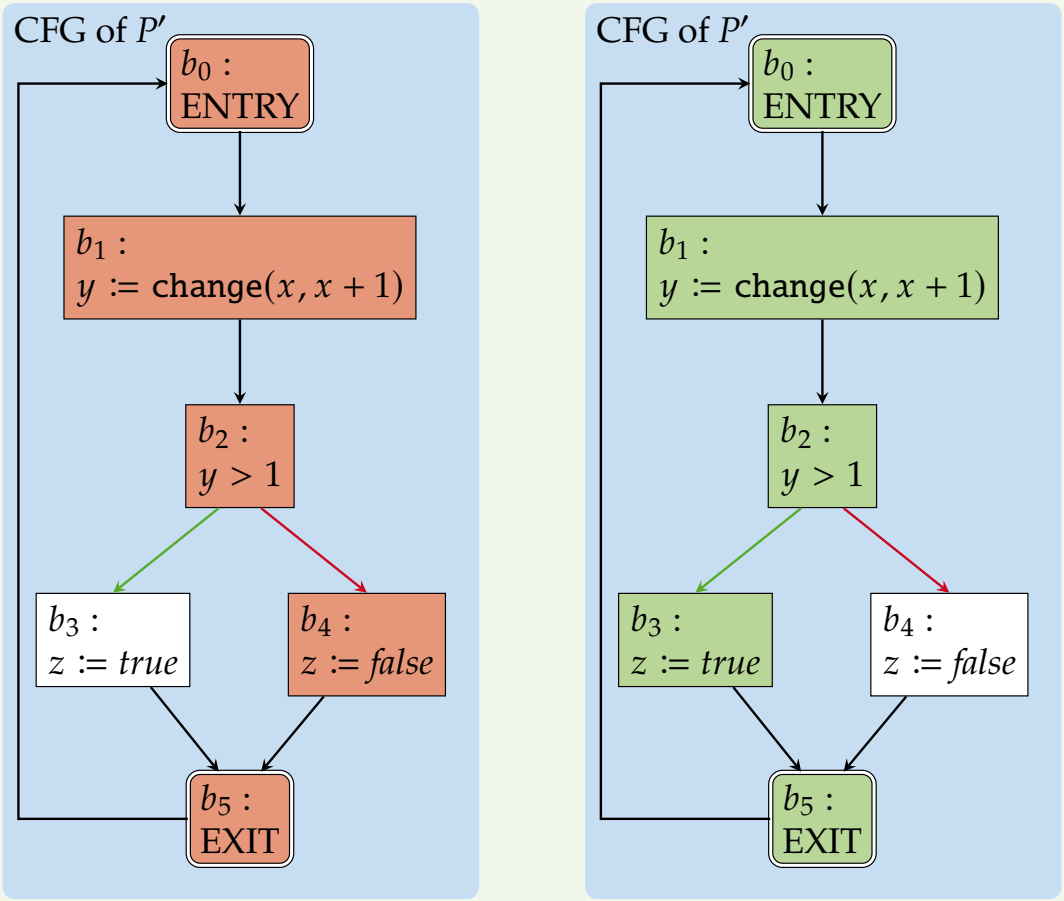
The four major steps of Algorithm 7 are explained throughout the following sections. If derivable from the context, the subscript of the divergent execution context $q_{divergent}$ is dropped in the following sections.

5.2.2 Collecting Change Traversing Test Cases

In Line 1 of Algorithm 7, the test suite T of the version before the reconfiguration is reused and executed on the CAP to determine which test cases are change-traversing. A test case is change-traversing if the execution path in the new program version induced by the valuations of the test case contains a change-annotated instruction. Before executing the new program version on the test case, the program's interface is checked. In case the test case does not contain valuations for all variables of the new program version's interface due to a reconfiguration, it is augmented. The test case is augmented with additional valuations using the 0-default initialization (see Section 2.1) for `BOOL` and `INT` as defined in the IEC 61131 standard [Int 14], *false* and 0, respectively.

Example 5.1: Collecting Change Traversing Test Cases

Given a test suite $T := \{ \underbrace{(x \mapsto 0, z \mapsto \mathit{false})}_{t_1}, \underbrace{(x \mapsto 2, z \mapsto \mathit{true})}_{t_2} \}$ generated for the old program P illustrated in Figure 5.4. Each test case $t_i \in T, i \in \{1, 2\}$ is "simulated" in the change-annotated CFG of the reconfigured program P' using the old expression of the `change`-annotation macro. The concrete execution of the test suite T yields a mapping from test cases to covered basic blocks.



(a) Traversed basic blocks after execution of t_1.

(b) Traversed basic blocks after execution of t_2.

For instance, t_1 traverses $\{b_0, b_1, b_2, b_4, b_5\}$ as the valuation $y \mapsto 0$ at the decision in b_2 yields *false* after concrete evaluation, whereas t_2 traverses $\{b_0, b_1, b_2, b_3, b_5\}$.

For each test case, the execution history and the state valuations reaching the end of the cycle in the old program versions are saved. This augmentation comes in handy later when test cases are checked for difference-revealing behavior between the old version and the reconfigured program. As a test case can traverse multiple change-annotated labels, only those test cases are selected that cover the most amount of change-annotated labels. This reduces the number of test cases needed for consideration in the first phase without losing expressiveness. Test cases spanning multiple cycles with the same valuations as prefixes are prioritized. There may exist no test case in the test suite T that covers a specific change-annotated label. This requires the generation of a test case covering that specific label before execution in case divergent behavior should be detected for that particular label.

The resulting subset of test cases $T_{change\text{-}traversing}$ of the test suite T forms the basis for the next phase.

5.2.3 Finding Divergent Execution Contexts

Finding divergent execution contexts establishes the basis for deriving test inputs that expose different behavior across both program versions. It is driven by the concrete valuations from the change-traversing test cases and executed for each test case touching a change-annotated label. Algorithm 8 shows the algorithm responsible for finding divergent execution contexts using SE.

Algorithm 8 findDivergentContexts

```
   Input  : CAP P = (G, 𝒢), Test case t_change-traversing
   Local  : Set of succeding execution contexts Q'
   Output : Set of divergent contexts and triggering test case D
 1: q ← initializeDivergenceExecution(G, t_change-traversing)
 2: for each ρ_t ∈ t_change-traversing do
 3:     ρ ← ρ[v ↦ ρ_t(v) | ∀v ∈ V_input]
 4:     CONTINUE_FINDING_DIVERGENT_CONTEXTS:
 5:     // Algorithm 9
 6:     (Q', ξ) ← executeCycleUntilDivergence(q)
 7:     switch ξ do
 8:         case EXPECTED_BEHAVIOR do
 9:             // continue execution with the next cycle
10:         end
11:         case POTENTIAL_DIVERGENT_BEHAVIOR do
12:             for each q'_forked ∈ Q' do
13:                 D.push(⟨q'_forked, deriveTestCase(q'_forked)⟩)
14:             end
15:             goto CONTINUE_FINDING_DIVERGENT_CONTEXTS
16:         end
17:         case DIVERGENT_BEHAVIOR do
18:             D.push(⟨q, t_change-traversing⟩)
19:             return D
20:         end
21:     end
22:     cycle ← cycle + 1
23: end
24: return D
```

Initially, an execution context is created, and internal data structures are populated. It is important to note that no merging occurs throughout the execution of the first phase, as divergent forked execution contexts are added to a queue to be explored in the second phase as shown in Algorithm 7. Hence in Line 3, execution

is driven by the concrete valuations of the test cases. The execution context is then executed until the end of the cycle or until a divergence has occurred. Before going in-depth with the algorithm in Line 6, the rest of the algorithm is explained first.

Expected Behavior In contrast to the SE in Algorithm 1 of Chapter 4, the engine also keeps track of the current execution status ξ. This execution status ξ denotes whether a divergence has occurred or not during the execution of the execution context q under the current change-traversing test cases in the respective cycle. If no divergence has been triggered and execution was performed as in traditional SE, the behavior of executing the context under the test case is as expected. In this case, the algorithm continues executing the next cycle regarding the valuations specified in the test case $t_{\textit{change-traversing}}$.

Potential Divergent Behavior Whenever potential divergent execution behavior is encountered during SE, it is examined by the engine in Lines 11 to 16. Suppose, for now, that the execution of Algorithm 9 returned a set of succeeding execution contexts Q' that exhibit potential divergent behavior. In that case, the forked divergent execution contexts are regarded as candidates that must be further explored in the second phase of Algorithm 7. Hence, they are added to the set of divergent contexts, and a test case that triggers the divergent behavior is derived. This test case is used during the BSE in the second phase of Algorithm 7 to systematically and comprehensively explore additional divergent contexts [Kuc & Pal+ 18]. Line 15 jumps back to finish executing the current execution context under the test case and potentially reveals a real divergence. As long as the concrete executions do not diverge, Algorithm 8 continues execution until the end of the test case and explores any other possible divergences along the execution path.

Divergent Behavior While the current change-traversing test case makes both program versions follow different paths at a branch instruction for an expression containing a "shadowed" variable, it might not be sufficient enough to explore all new divergent behaviors that arise from this context. The execution context q is pushed together with the change-traversing test case to the set of divergences found during the first phase, and execution is stopped for the current test case.

In the following, the algorithms responsible for detecting the divergent behavior in Line 6 of Algorithm 8 and the core concept are explained in detail.

Algorithm 9 executeCycleUntilDivergence

Input : Execution context q
Output : Execution status ξ, Set of succeeding execution contexts Q'

```
1: while ¬reachedSucceedingCycle(q) do
2:     // Algorithm 10
3:     (ξ, Q' := {q'_1, ..., q'_n}) ← executeVertexUntilDivergence(q)
4:     if ξ ≠ EXPECTED_BEHAVIOR then
```

```
5:      | return (ξ, Q′)
6:    end
7:  end
8:  return (EXPECTED_BEHAVIOR, Q′ := {q})
```

The execution context is executed until either a divergence is encountered (see Line 4 of Algorithm 9) or the end of the current cycle has been reached (see Line 1 of Algorithm 9). The exploration strategy of SSE is similar to that of the BSE presented in Chapter 4. Algorithm 10 and Algorithm 11 are explained in the following.

Algorithm 10 executeVertexUntilDivergence

Input : Execution context $q := (c, s = (b_\ell, \rho, \sigma, \pi), C)$
Output : Execution status ξ, Set of succeeding execution contexts Q'

```
1:  switch handleVertex(b_ℓ) do
2:    case PROGRAM_ENTRY do
3:      ξ ← EXPECTED_BEHAVIOR
4:      b_ℓ′ ← getSucceedingVertex(b_ℓ)
5:      s′ ← (b_ℓ′, ρ, σ, π)
6:      Q′.push(q′ := (c, s′, C))
7:    end
8:    case FUNCTION_BLOCK_ENTRY do
9:      // analogous to PROGRAM_ENTRY
10:     ξ ← EXPECTED_BEHAVIOR
11:     b_ℓ′ ← getSucceedingVertex(b_ℓ)
12:     s′ ← (b_ℓ′, ρ, σ, π)
13:     Q′.push(q′ := (c, s′, C))
14:   end
15:   case REGULAR do
16:     // Algorithm 11
17:     (ξ, Q′) ← executeInstructionUntilDivergence(q)
18:   end
19:   case PROGRAM_EXIT do
20:     ξ ← EXPECTED_BEHAVIOR
21:     // concrete store ρ is overriden by change-traversing test case
22:     σ′ ← σ[v ↦ v_fresh | v ∈ V_input]
23:     s′ ← (b_ℓ′, ρ, σ′, π)
24:     Q′.push(q′ := (c + 1, s′, C))
25:   end
26:   case FUNCTION_BLOCK_EXIT do
27:     ξ ← EXPECTED_BEHAVIOR
28:     (_, _, b_ℓ_return) ← C.top()
29:     C′ ← C.pop()
30:     s′ ← (b_ℓ_return, ρ, σ, π)
```

```
31:     |   Q'.push(q' := (c, s', C'))
32:     end
33: end
34: return (ξ, Q')
```

Algorithm 10 is mostly analogous to Algorithm 3 of Chapter 4. The only differences lie in the propagation of the execution status ξ, the specially tailored execution of instructions in Line 17, and the missing update of concrete valuations in case the program exit has been reached. This is because the valuations from the concrete store are overridden by the valuations of the change-traversing test case driving the execution in Line 2 of Algorithm 8.

In the following, Algorithm 11 is explained in further depth as it is the heart of finding divergent execution contexts in SSE.

Algorithm 11 executeInstructionUntilDivergence

```
   Input  : Execution context q
   Output : Execution status ξ, Set of succeeding execution contexts Q'
 1: switch instructionAt(b_ℓ) do
 2:     case v := e do
 3:         b_ℓ' ← getSucceedingVertex(b_ℓ)
 4:         (ρ', ρ'_s) ← ρ[v ↦ eval_ρ^shadow(e)]
 5:         (σ', σ'_s) ← σ[v ↦ eval_σ^shadow(e)]
 6:         q' ← (c, s' = ((b_ℓ', ρ', σ', π), ρ'_s, σ'_s, π_old, π_new), C)
 7:         Q'.push(q')
 8:     end
 9:     case b do
10:         if containsShadowExpression(b) then
11:             (d_old, d_new) ← eval_ρ^shadow(b)
12:             if d_old ≠ d_new then
13:                 return (DIVERGENT_BEHAVIOR, Q' := {q})
14:             else
15:                 (φ_old, φ_new) ← eval_σ^shadow(b)
16:                 b_ℓ,true ← getSucceedingPositiveVertex(b_ℓ)
17:                 b_ℓ,false ← getSucceedingNegativeVertex(b_ℓ)
18:                 if tryDivergentFork(π ∧ π_old ∧ π_new ∧ ¬φ_old ∧ φ_new) then
19:                     ρ' ← model(π ∧ π_old ∧ π_new ∧ ¬φ_old ∧ φ_new)
20:                     q' ← (c, s' = (b_ℓ,true, ρ', σ, π_old ∧ ¬φ_old, π_new ∧ φ_new), C)
21:                     Q'.push(q')
22:                 end
23:                 if tryDivergentFork(π ∧ π_old ∧ π_new ∧ φ_old ∧ ¬φ_new) then
24:                     ρ' ← model(π ∧ π_old ∧ π_new ∧ φ_old ∧ ¬φ_new)
25:                     q' ← (c, s' = ((b_ℓ,false, ρ', σ, π), ρ_s, σ_s,
                                π_old ∧ φ_old, π_new ∧ ¬φ_new), C)
26:                     Q'.push(q')
```

```
27:                 end
28:                 if d_old then
29:                     q' ← (c, s' = (b_ℓ,true, ρ, σ, π_old ∧ φ_old, π_new ∧ φ_new), C)
30:                     Q'.push(q')
31:                 else
32:                     q' ← (c, s' = (b_ℓ,false, ρ, σ, π_old ∧ ¬φ_old, π_new ∧ ¬φ_new), C)
33:                     Q'.push(q')
34:                 end
35:                 return (POTENTIAL_DIVERGENT_BEHAVIOR, Q')
36:             end
37:         else
38:             // analogous to Algorithm 4
39:         end
40:     end
41:     case G() do
42:         // analogous to Algorithm 4 modulo summary application
43:     end
44: end
45: return (EXPECTED_BEHAVIOR, Q')
```

Unlike traditional SE, the execution of assignments may introduce shadow expressions. Lines 2 to 7 of Algorithm 11 concretely evaluate and symbolically encode the expression of the assignment and introduce a new concrete and symbolic shadow variable in case the expression contains a change-annotation expression `change(old:` e_1`, new:` e_2`)`. This is an essential part for SSE as it gives rise to efficiently sharing the execution states using those auxiliary shadow variables. An example is given in Example 5.2 to depict how concrete and symbolic representations of changes are reflected.

Example 5.2: Concrete and Symbolic Representations of Changes

Consider the assignment $x := x + \texttt{change}(y, z)$ under the current symbolic store $\sigma := \{x_0 \mapsto 0, y_0 \mapsto y_0, z_0 \mapsto z_0\}$. The symbolic expression tree for the valuation of x after the execution is shown in Figure 5.8.

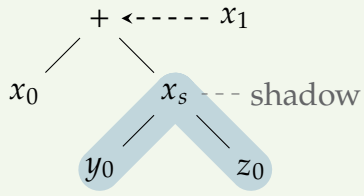


Figure 5.8: Symbolic expression tree containing a shadow expression.

An auxiliary symbolic shadow variable x_s is introduced and saved in the symbolic shadow store $\sigma_s := \{x_s \mapsto (\texttt{old:}\ y_0, \texttt{new:}\ z_0)\}$. The symbolic store

$\sigma := \{x_1 \mapsto x_0 + x_s, \ldots\}$ uses the symbolic shadow variable x_s to "shadow" the current symbolic valuation of the second operand of the assignment. Depending whether the old or the new program version is regarded, the symbolic valuation of x_s is substituted respectively in the corresponding $\text{eval}_\sigma^{shadow}$ function. The case for the concrete evaluation is analogous.

During TSA, the branching instructions are handled in a particular way in Lines 9 to 40 of Algorithm 10 different from the one presented in Algorithm 4 of Chapter 4.

As `change` annotations may occur in nested expressions, Line 10 of Algorithm 11 checks recursively whether the current branching expression contains a shadow expression. If the expression does not contain a shadow expression, execution continues with Algorithm 4 presented in Chapter 4. If it contains a shadow expression, it may influence the branching behavior and needs to be checked respectively. Example 5.3 covers the handling of branching instructions in the running example.

Example 5.3: Finding Divergent Contexts

Consider that the test case $t_1 := (x \mapsto 0, z \mapsto \mathit{false})$ is executed on the reconfigured CFG of the program depicted in Figure 5.4.

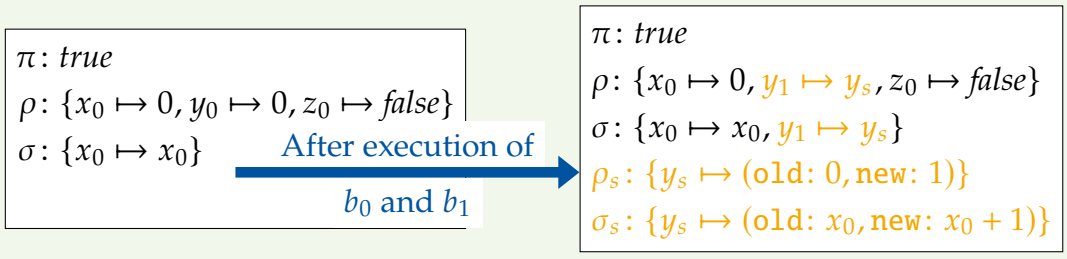


Figure 5.9: Initial execution context and the effect of execution after reaching basic block b_2.

The evolution of the initial execution context after executing b_0 and b_1 is depicted in Figure 5.9. While the execution of the entry basic block b_0 did not affect the initial execution context, the execution of the change-annotated basic block b_1, introduced a concrete and symbolic change-shadow expression y_s. The execution of the branch instruction $y_1 > 1$ of b_2 may lead to either a

1. divergence,
2. potential divergence,
3. or no divergence,

} AST contains `change(old, new)`

depending on the symbolic representation of y_1. This is checked in Algorithm 11 by calling the `containsShadowExpression`-function on the expression of the branching instruction. It checks, whether the Ast of $y_1 > 1$ under the given execution context contains a shadowed expression.

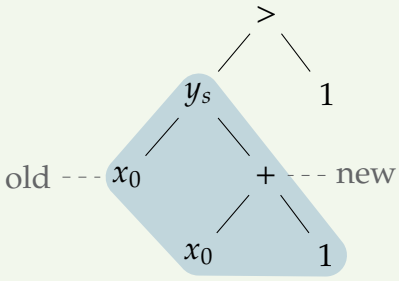


Figure 5.10: Ast of $y_1 > 1$ under the current execution context.

As Figure 5.10 graphically illustrates the Ast of the execution context from Figure 5.9 containing a shadow expression, the execution **may diverge** and need to be further analyzed.

Line 11 evaluates the branch expression under both the concrete and the shadowed concrete store, resolving all shadow expressions [Gro & Völ+ 22b]. Example 5.4 shows how the check in Line 12 of Algorithm 11 is applied to the running example.

Example 5.4: Checking for Divergences

When checking for divergences, the branching expression is evaluated concretely under the current execution context. Consider the following execution context obtained after the execution of the initial execution context on the CFG of the running example as shown in Example 5.3:

$$\begin{aligned}
&\pi : \mathit{true} \\
&\rho : \{x_0 \mapsto 0, y_1 \mapsto y_s, z_0 \mapsto \mathit{false}\} \\
&\sigma : \{x_0 \mapsto x_0, y_1 \mapsto y_s\} \\
&\rho_s : \{y_s \mapsto (\texttt{old}: 0, \texttt{new}: 1)\} \\
&\sigma_s : \{y_s \mapsto (\texttt{old}: x_0, \texttt{new}: x_0 + 1)\}
\end{aligned}$$

Concrete evaluation of the branching expression $y_1 > 1$ under this context as shown in Line 11 of Algorithm 11 yields the following concrete values for

d_{old} and d_{new}:

$$d_{old} = \texttt{eval}_{\rho}^{shadow}(y_1 > 1) = \texttt{eval}_{\rho}^{shadow}(0 > 1) = \mathit{false}$$
$$d_{new} = \texttt{eval}_{\rho}^{shadow}(y_1 > 1) = \texttt{eval}_{\rho}^{shadow}(1 > 1) = \mathit{false}$$

A subsequent check for equality of $d_{old} = d_{new}$ yields *true* and hence the concrete execution do not diverge.

If the valuations of the expressions under the old and the new execution context do not coincide, the seeded test case leads to divergent behavior and might trigger difference-revealing outputs [Gro & Völ+ 22b] giving rise to the concept of *four-way* forking.

Four-way Forking of Execution Paths

Figure 5.11 illustrates the concept of *four-way* forking [Pal & Kuc+ 16; Kuc & Pal+ 18]. Whenever the concrete execution evaluates the branching expression to the

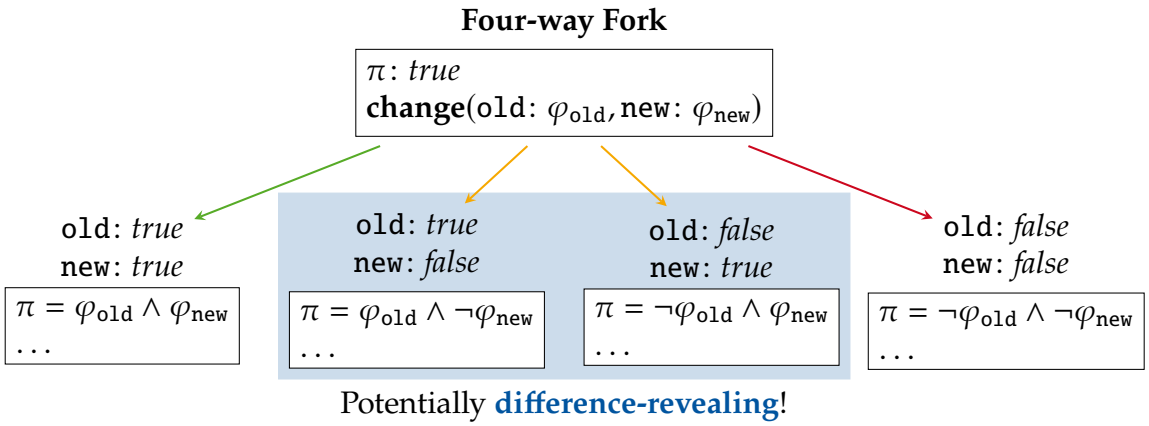


Figure 5.11: Four-way forking in SSE.
Figure adapted from Figure 2 in [Pal & Kuc+ 16].

same value, both program versions make the same decision. This is denoted by the two outer branches of Figure 5.11, where `old` and `new` are both *true* and *false*, respectively. In case the branching expression evaluates to different values under the current execution contexts, potential divergent behavior may occur. Through the use of the shadow expression, the forking of the execution context is not limited to the traditional fork into two succeeding execution contexts for the respective *true* and *false* branches. This is represented in Lines 18 to 27 of Algorithm 11, where the generation of divergent execution contexts can be performed. Example 5.5 illustrates this process for the running example.

Example 5.5: Divergent Fork

After concluding that the concrete evaluations of the branching expression $y_1 > 1$ for the running example do not diverge, Algorithm 11 tries to fork the current execution context in Lines 18 to 27 symbolically and divergently. For the sake of example, the execution context below is considered:

$$\begin{array}{l} \pi : \mathit{true} \\ \rho : \{x_0 \mapsto 0, y_1 \mapsto y_s, z_0 \mapsto \mathit{false}\} \\ \sigma : \{x_0 \mapsto x_0, y_1 \mapsto y_s\} \\ \rho_s : \{y_s \mapsto (\texttt{old}: 0, \texttt{new}: 1)\} \\ \sigma_s : \{y_s \mapsto (\texttt{old}: x_0, \texttt{new}: x_0 + 1)\} \end{array}$$

First, the branching expression $y_1 > 1$ is evaluated under the symbolic shadow store σ_s using the $\texttt{eval}_{\sigma}^{shadow}$ function:

$$\varphi_{old} = \texttt{eval}_{\sigma}^{shadow}(y_1 > 1) = x_0 > 1$$
$$\varphi_{new} = \texttt{eval}_{\sigma}^{shadow}(y_1 > 1) = x_0 + 1 > 1$$

Next, it is checked whether $\texttt{tryDivergentFork}(\varphi_{\texttt{old}} \wedge \neg\varphi_{\texttt{new}})$ is **SAT**, while for the sake of example, the old, new, and current path constraints are omitted from the query:

$$\underbrace{(x_0 > 1)}_{=\varphi_{\texttt{old}}} \wedge \underbrace{\neg(x_0 + 1 > 1)}_{=\neg\varphi_{\texttt{new}}} \equiv (x_0 > 1) \wedge (x_0 \leq 0) \text{ is } \textbf{UNSAT}!$$

As the assertions are unsatisfiable, no divergence toward the left-innermost branch of the four-way fork depicted in Figure 5.11 is possible. Second, it is checked whether $\texttt{tryDivergentFork}(\neg\varphi_{\texttt{old}} \wedge \varphi_{\texttt{new}})$ is **SAT**:

$$\underbrace{\neg(x_0 > 1)}_{=\neg\varphi_{\texttt{old}}} \wedge \underbrace{(x_0 + 1 > 1)}_{=\varphi_{\texttt{new}}} \equiv (x_0 \leq 1) \wedge (x_0 > 0) \text{ is } \textbf{SAT}, \text{ witness } x_0 = 1!$$

This set of assertions is satisfiable and the underlying SMT solver returns a model from which the concrete valuations driving the executions into this particular branch are extracted.

The execution context q' reaching the branching point is forked and updated with the retrieved model from the solver. It is added to the queue of divergent executions contexts Q' for exploration in the subsequent phase by appropriately propagating it to Lines 11 to 16 of Algorithm 8. Test inputs derived from divergent execution

contexts drive the SE in one of the inner branches of Figure 5.11 and belong to the class of non-obsolete, modification-traversing test cases (cf. Figure 1.13). In Lines 28 to 34 of Algorithm 11 the current execution context is updated to follow the path of the branch adhering to the respective decision value of d_{old}. This ensures that deeper nested divergences may be found in case this branch did not introduce any divergent execution contexts in the prior Lines 18 to 27.

The first two steps of Figure 5.6 identify whether there exist modification-traversing test cases in the test suite T and derives new test cases that trigger divergent behavior. The next step investigates whether the modification-traversing test cases found during the evaluation of the four-way fork depicted in Figure 5.11 propagate other modification- and difference-revealing behavior.

5.2.4 Propagating Divergent Execution Contexts

The second phase consists of Lines 5 to 8 of Algorithm 7 and initiates a BSE for each divergent context found in the first phase. While in the original implementation, the BSE is executed in a breadth-first search mode for a fixed time budget [Kuc & Pal+ 18], the algorithm of this thesis uses a cycle-based, depth-first search. Essentially, this type of depth-first search simulates a breadth-first search throughout one execution cycle of the PLC, as execution contexts are executed until the end of the cycle, where the output behavior is observable. This also benefits test case generation, leading to concise test cases without unnecessary cycles in between. The BSE is initialized with the test case t', which triggered the divergent behavior. At this point, the behavior triggered by that test case is either a "real" divergence as the concrete executions in the old and the new program version deviated, or it was generated because of a potential, possible divergence at the four-way fork. Either way, the BSE starts from the divergence point inheriting the execution context, which triggered the divergence to search for additional divergent behaviors [Kuc & Pal+ 18]. As this search is performed in the new version of the `change`-annotated program, only the common and the "new" path constraint are considered. Therefore, the merging strategy from Section 4.1.1 does not need to be modified to account for both program versions.

Other than that, this phase behaves as the SE presented in Chapter 4, generating as many test cases as possible until the respective termination criteria are met [Gro & Völ+ 22b]. Paths covered by these test cases originate from a divergent execution context and hence may be difference-revealing. This approach is, however, incomplete [Nol & Ngu+ 19] as the four-way fork is not executed exhaustively and hence does not partition the entire input space with regards to the divergence. Nevertheless, the aim of SSE is not to generate a high-coverage test suite [Kuc & Pal+ 18] but rather to generate test inputs that exercise the divergences.

5.2.5 Checking for Output Differences

Lastly, the old and new program versions are checked for output differences in Line 8 of Algorithm 7 on the test cases exposed during the prior phases to be triggering divergent execution contexts. For each divergent test case, the new program version P' is simulated with the valuations of that test case. If the outputs of the test case in the old program version P and the new program version P' differ, the test case is difference-revealing and added to the set of difference-revealing test cases. The following Example 5.6 illustrates this process on the running example.

Example 5.6: Checking for Output Differences

After the derivation of test inputs that lead to divergent execution contexts, they are executed to check whether they propagate the divergence and reveal differences in the outputs of both program versions. By concretely executing the derived divergent test input $x_0 = 1$ in the change-annotated CFG of the program, the following execution contexts are obtained:

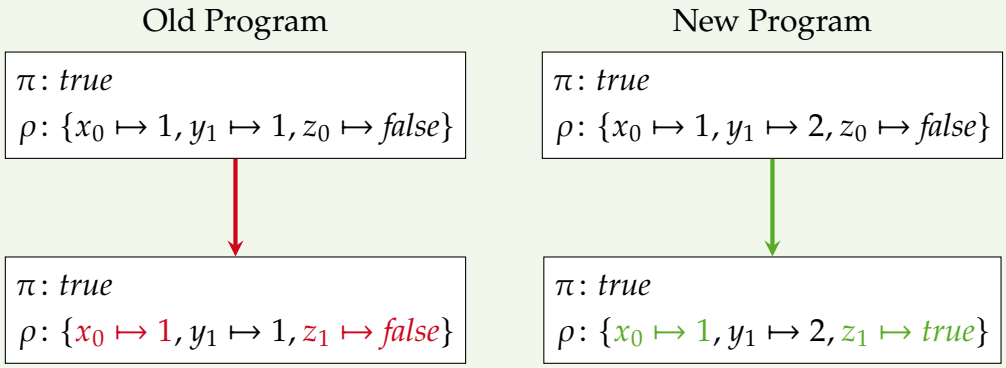


Figure 5.12: Resulting execution contexts after concretely executing the old and the new program on the derived divergent test input.

As both program versions produce different outputs (`old`: $z_1 \mapsto \mathit{false}$ vs. `new`: $z_1 \mapsto \mathit{true}$) for the same input (`old` and `new`: $x_0 \mapsto 1$) the test input is said to be **difference-revealing**.

As illustrated in Figure 5.5, it is the task of the developer to check whether a difference-revealing test case is a fault-revealing test case or not.

Concluding Remarks This chapter presented a state-of-the-art algorithm for TSA lifted to the domain of PLC software. The necessary adaptations and the embedding of the CSE of Chapter 4 have been presented. The next chapter evaluates the performance of the proposed TSA algorithm.

6 Evaluation

This chapter first introduces and provides background information on the benchmarks used to evaluate this thesis. The actual evaluation is two-fold. Both, the contribution of Chapter 4 and the contribution of Chapter 5 are evaluated in this order, and the results are discussed.

Technical Setup The evaluation was conducted on an Intel(R) Core(TM) i5-6600K CPU @ 3.50 GHz x 4 desktop with 16 GiB of RAM running Ubuntu 22.04.1 LTS. The high-performance automated theorem prover Z3 version 4.9.1 by Microsoft [dMou & Bjø 08] was used for SMT-solving. The benchmarks evaluated with ARCADE.PLC were also run with the same evaluation setup.

6.1 Benchmarks

Due to the lack of industrial-sized benchmarks that incorporate reconfigurations and yet are still analyzable by the implemented proof of concept, the PLCopen Safety suite and the PPU have been chosen as a compromise.

6.1.1 PLCopen Safety Suite

The PLCopen Safety suite consists of a set of IEC 61131-3 programs [Int 14] and FBs defined by the members of the PLCopen and external safety-related organizations. PLCopen is an independent worldwide organization aiming to provide efficiency in industrial automation based on the needs of its users[1]. They concentrate on technical and vendor-neutral specifications around the IEC 61131-3 standard to reduce the costs of reimplementation and validation of FBs. An exemplary PLC program is depicted in Figure 6.1. It is described in the technical specification by the PLCopen [PLC 08]. The description consists of a textual specification of selected important properties, a visualization of the behavior using digital timing diagrams, and a semi-formal specification of the complete behavior in the form of state diagrams [Bia 16]. The implementations for the rest of the PLCopen Safety suite [PLC 18] were reused from the ARCADE.PLC tool [Bia & Bra+ 12; Bia 16].

[1] https://plcopen.org/what-plcopen

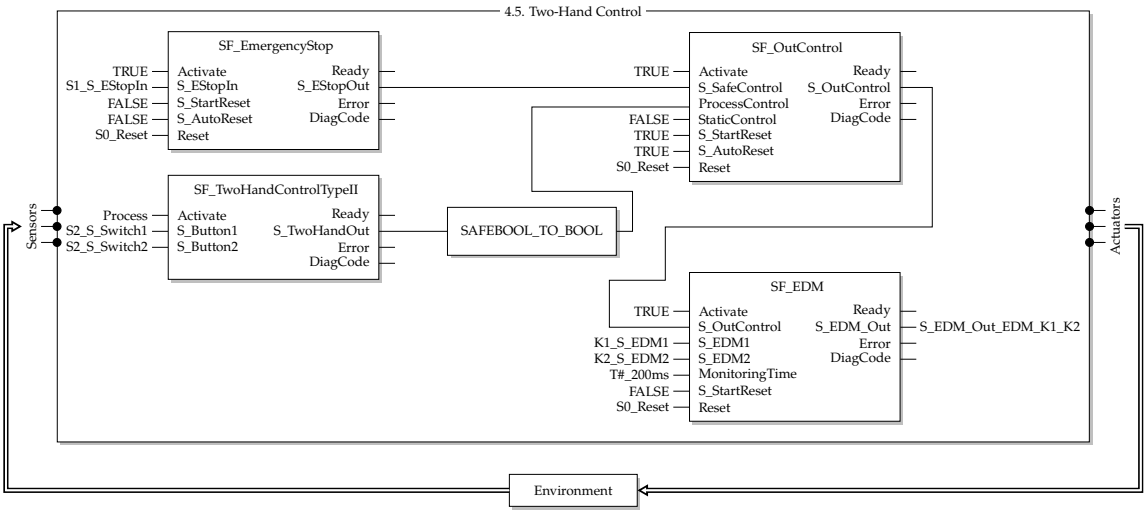


Figure 6.1: An exemplary PLC program interacting with its environment.
Figure adapted from Figure 23 in [PLC 08].

6.1.2 Pick and Place Unit

The PPU is a bench-scale manufacturing system [Vog & Leg+ 14]. In total, 15 scenarios and their evolutions are described in the technical document. It is an open case study, and the documentation is freely available[2]. Its size and complexity limitations pose a trade-off between problem complexity and evaluation effort and hence make it suitable for algorithmic analysis.

While it was extended to feature 23 scenarios [Vog & Bou+ 18], the benchmarks analyzed in this thesis are limited to the original description of the PPU [Vog & Leg+ 14]. Table 6.1 gives an overview of the software and mechanical changes

Table 6.1: Software and mechanical changes during evolution of the PPU.
The symbols are explained in the text.
Table adapted from Table 20 in [Vog & Leg+ 14].

Scenario	Stack		Crane		Ramp		Stamp	
	Soft.	Mech.	Soft.	Mech.	Soft.	Mech.	Soft.	Mech.
`Scenario_0`	I	I	I	I	-	I	-	-
`Scenario_1`	=	=	=	=	-	M	-	-
`Scenario_2`	M	M	=	=	-	=	-	-
`Scenario_3`	=	=	M	M	-	=	I	I

during the evolution of the PPU. A graphical representation of Table 6.1 is given in Figure 6.2. `Scenario_0` introduces (I) the stack, crane, and ramp components.

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

The evolution from `Scenario_0` to `Scenario_1` introduces a purely structural

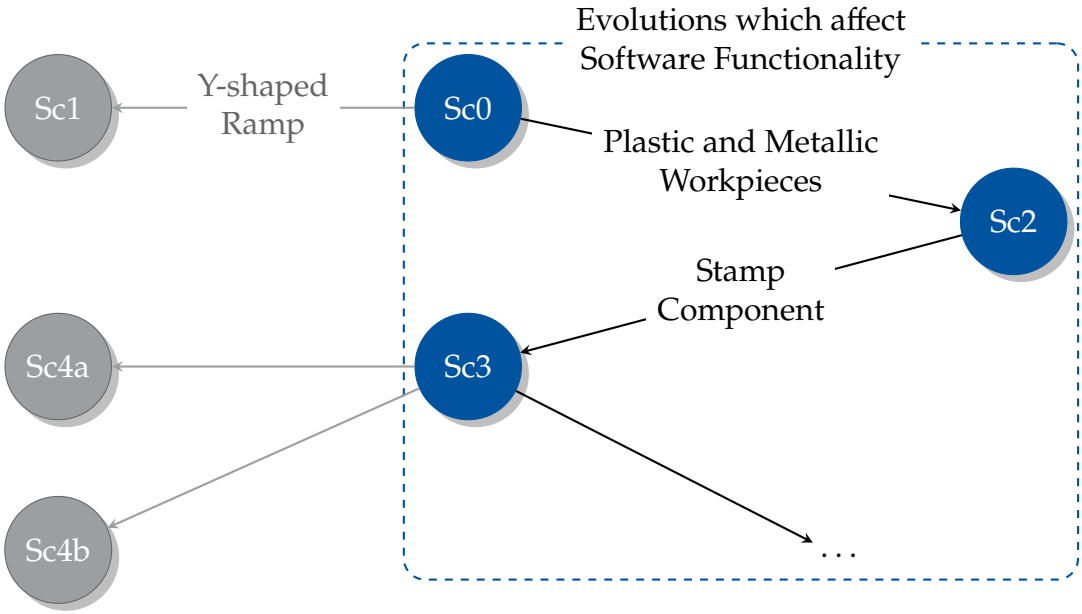


Figure 6.2: Excerpt from evolutions of the PPU.
Figure adapted from Fig. 48 in [Vog & Leg+ 14].

change in the form of a modification (**M**). The ramp component underwent a mechanical reconfiguration which did not affect the software, and all the other components were untouched by this reconfiguration (=).

The second evolution step from `Scenario_0` to `Scenario_2`, as depicted in Figure 6.2, resulted from changing customer requirements. The PPU should handle both plastic and metallic workpieces. Due to the introduction of an induction sensor (structural change), the output behavior of the stack (functional change) changed while all other components were untouched by this reconfiguration.

The third evolution step from `Scenario_2` to `Scenario_3` was performed in order to stamp metallic workpieces before being transported to the ramp. A stamp component was introduced to enable stamping, and the behavior of the crane had to be reimplemented.

Figure 6.3 illustrates an exemplary scenario of the PPU. The PPU is designed for a centralized control structure executing multiple FBs on the same PLC. While this centralized control structure is certainly capable of orchestrating a whole work cell, the communication between the components is still limited to the scope of within one application. Therefore, the TSG of this thesis is characterized as classic, monolithic testing rather than distributed testing and thus challenges arising from diverse networking between components are not regarded.

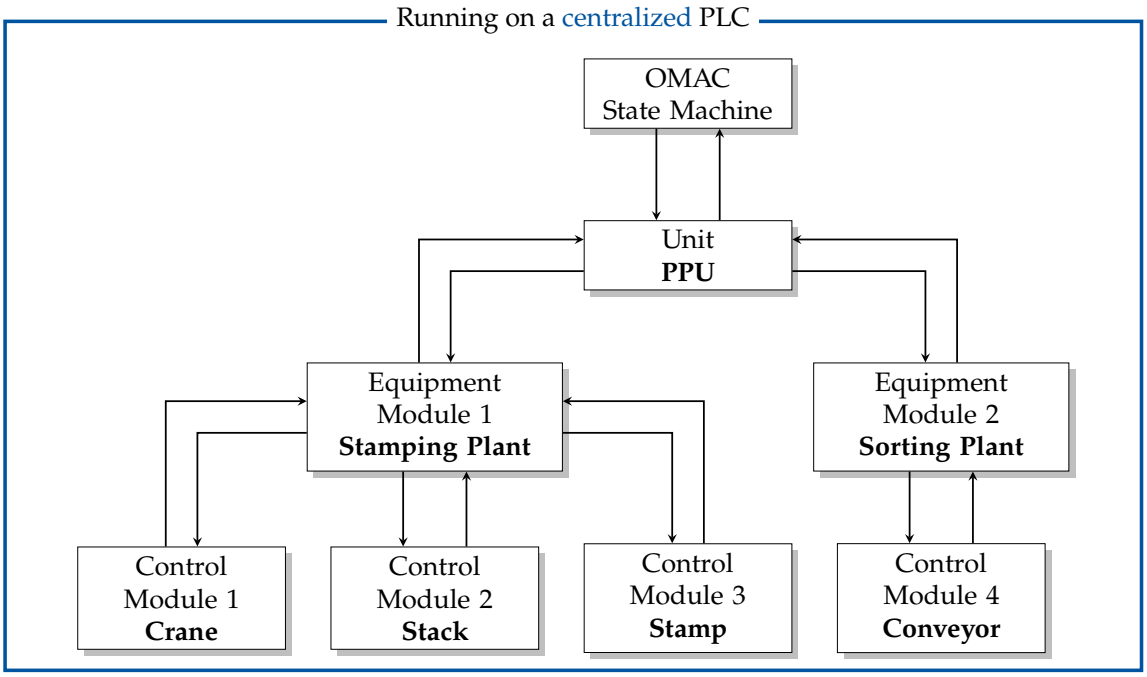


Figure 6.3: An exemplary scenario of the PPU.
Figure adapted from Figure 17 in [Vog & Rös+ 16].

Setup The VerifAps library[3] was used to translate the scenarios of the PPU from PLCopenXML to ST. The Xml2TxtApp of VerifAps was used with the options `-translate-sfc` and `-ppu` and manually post-processed to conform with the input defined by the grammar used by the PAF for this thesis.

6.2 Test Suite Generation

As TSG is an integral part of TSA, first, the achieved performance improvements for the BSE are presented.

Evaluation of the PLCopen Safety Suite Table 6.2 shows the results of the evaluation of a set of safety-related PLC programs provided by the PLCopen organization [PLC 18] using their corresponding reference implementation in Arcade.PLC [Bia & Bra+ 12] which were published in [Gro & Völ+ 22b]. For each evaluated FB and user-defined program, Table 6.2 shows the lines of code (**LOC**), the achieved branch coverage values (**cov. [%]**) along multiple execution cycles, the amount of generated test cases (**T [#]**), as well as the runtimes (**time [s]**) in seconds of the merge-based TSG algorithm in Arcade.PLC [Boh & Sim+ 16] in

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

Table 6.2: Comparison of branch coverage and runtimes for the TSG of the PLCopen Safety library, ordered alphabetically. The rows highlighted in blue show results in favor of ARCADE.PLC and the rows highlighted in orange show results where the contribution significantly outperformed ARCADE.PLC in one metric.

Table adapted from Table 1 in [Gro & Völ+ 22b].

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

comparison to the results of this thesis' contribution. The timeout (TO) was set to 10 minutes, and the values show an average over multiple experiments.

ARCADE.PLC uses a value-set analysis for the detection of unreachable branches, whereas this contribution uses the SA capabilities provided by CRAB. To only focus on the performance of the underlying BSE implementations, both programs were executed with the additional pre-computed information from the SA without considering its execution time. SA_{manual} refers to the manual annotation for truly unreachable branches in addition to the derived automated results from CRAB, which were over-approximated due to the convexification of the disjunctions as mentioned in Section 4.1.4.

Overall, both approaches perform equally well concerning the analysis time of the corresponding FBs. There are a few outliers (`Enable_Switch`, `Mode_Selector`, `Muting_Seq`, `Safely_Limit_Speed`, and `Testable_Safety_Sensor`), but in general, the results are as expected when taking the **LOC** as a reference for a rough estimate of the complexity. A significant difference is visible in the number of tests generated by both approaches. While ARCADE.PLC generates concise test cases for every branch, the contribution of this thesis tries to avoid redundancies due to shorter test cases being a prefix of longer test cases, hence generating fewer test cases overall [Gro & Völ+ 22b]. This observation is neither a benefit nor a disadvantage of either approach and could be easily obtained by static post-processing on the test suite generated by ARCADE.PLC. Due to a technical limitation, ARCADE.PLC does not generate any test cases when running into a timeout.

Regardless, the coverage values can be used as a measure of the achieved performance. A trend visible during the analysis of `Muting_Seq`, which passes onto the analysis of the user-defined programs, is that ARCADE.PLC quickly reaches its limitation when encountering hard-to-analyze and deeper nested explorations.

The user-defined programs found in the bottom section of Table 6.2 are composed of multiple FBs from the upper part of the table with additional logic and were analyzed by ARCADE.PLC with SA and by this contribution without manual SA annotations.

An apparent result derived from the achieved coverage values is that the analysis becomes more complex as more calling contexts are available. The delayed merge strategy until the end of the cycle, as performed by ARCADE.PLC, performs worse than merging on all realizable paths as soon as the opportunity is given [Gro & Völ+ 22b]. The degeneration of performance is most notably in programs that make heavy use of the timer and edge trigger FBs as deeper nested behavior can only be reached traversing specific paths requiring the heavy lifting of the underlying SMT solver.

Evaluation of the PPU Table 6.3 shows the analysis results of selected scenarios from the PPU benchmark. As TSA uses a BSE at its core, the results of the TSG without SA information are shown in Table 6.3 and discussed next. The semantics of the columns is the same as in Table 6.2 with the addition of the maximum

Table 6.3: Results of the TSG using BSE for selected PPU scenarios.
Table adapted from Table 2 in [Gro & Völ+ 22b].

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

number of cycles (**cycle [#]**). The BSE is configured to be bounded by time (TO = 30 minutes), the number of cycles (= 25), and reaching a predefined coverage value (= 100% "branch" coverage). The PPU has more functionality with regard to the FBs of the PLCopen Safety suite analyzed before, which is also reflected in the required time and the need to specify additional termination criteria for the TSG. A comparison with ARCADE.PLC was omitted as it is missing functionality to be able to analyze the PPU benchmark.

As the reconfiguration from `Scenario_0` to `Scenario_1` aims to increase the ramp's capacity by a purely structural change, as depicted in Table 6.1, the achieved results are equal in both cases. The differing runtimes in both rows are due to the average of multiple experiments. The other entries are as expected: as the size and complexity of the benchmark increase, the runtimes and the number of test cases also increase.

Using Summarization during Test Suite Generation

The summarization algorithm presented in Chapter 4 and the use of summaries during TSG were evaluated on the PLCopen Safety suite. Table 6.4 shows the results of the CSE without SA information and the CSE using summarization without SA information on the PLCopen Safety suite. The timeout was set to 10 minutes, and the execution was bounded by 15 cycles. For each evaluated FB, Table 6.4 shows the achieved coverage in percent (**cov. [%]**), the number of generated test cases (**T [#]**), as well as the runtimes (**t[s]**) in seconds of both implementations. In addition, the columns for the evaluation with summarization also show the number of generated summaries (**S [#]**) and the time it took to generate them (**gen. t[s]**). The overall runtime (**t[s]**) in seconds includes the cumulative time it took to apply the summaries during CSE (**apl. t[s]**) [Gro & Völ+ 22a].

When glancing at Table 6.4, it becomes apparent that both techniques perform more or less equally well. This was not expected. However, the time for summary application is already an indicator of the bottleneck, and subsequent profiling of the software architecture resulted in SMT-solving being the most expensive task, occupying around 67% of the total runtime.

Table 6.4: Comparison of branch coverage and runtimes for the TSG of the PLCopen Safety library. The rows highlighted in blue denote results where the use of the summarization heuristic performs better with regard to the highlighted metrics.
Table adapted from Table I in [Gro & Völ+ 22a].

Function Block / Program	CSE w/o SA			CSE + Sum. w/o SA			
	cov. [%]	**T [#]**	**t [s]**	**cov. [%]**	**T / S [#]**	**gen. t [s]**	**apl. t [s] / t [s]**
`Antivalent`	80.26	21	10.47	80.26	21 / 36	0.18	12.91 / 13.79
`EDM`	97.86	64	10.99	97.86	80 / 65	0.56	12.88 / 13.78
`Emergency_Stop`	94.29	26	9.75	94.29	**24 / 25**	0.15	**8.23 / 9.07**
`Enable_Switch`	93.42	31	12.97	93.42	34 / 32	0.14	13.13 / 14.40
`Equivalent`	93.75	24	8.95	93.75	27 / 34	0.13	12.80 / 13.77
`ESPE`	94.25	25	10.14	94.25	**24 / 25**	0.15	**8.35 / 9.15**
`Guard_Locking`	97.56	38	16.60	97.56	45 / 36	0.20	25.38 / 27.40
`Guard_Monitoring`	93.02	34	19.07	93.02	41 / 40	0.20	22.52 / 24.43
`Mode_Selector`	94.44	29	14.67	**95.83**	61 / 59	0.68	17.33 / 18.48
`Muting_Seq`	95.90	54	56.66	95.90	69 / 56	0.46	93.94 / 103.25
`Out_Control`	95.71	31	11.64	95.71	37 / 31	0.15	15.46 / 16.88
`Safe_Stop`	94.88	31	9.58	94.88	37 / 93	0.60	20.74 / 21.77
`Limit_Speed`	97.87	41	24.38	97.87	55 / 121	0.79	35.75 / 37.27
`Safety_Request`	95.56	36	15.99	95.56	**35 / 36**	0.29	17.78 / 19.63
`Safety_Sensor`	91.67	69	14.49	91.67	80 / 67	0.63	**12.49 / 13.55**
`TH_Control_T_II`	96.51	38	18.33	96.51	40 / 39	0.16	18.52 / 19.91
`TH_Control_T_III`	96.36	46	33.34	96.36	48 / 53	0.32	45.15 / 48.25

This shows that there is no clear benefit of using summarization over the baseline, and often, more test cases were generated due to the loss of context. At the same time, both analyses reached comparable coverage values. These test cases are redundant with regard to the chosen branch coverage metric as they represent witnesses of the same equivalence classes [Gro & Völ+ 22a].

Nonetheless, there are some cases where summarization outperforms the baseline. In general, no clear evidence could be derived on why because it staggers between FBs of varying complexity. One explanation could be that summarization covers multiple branches in one path and hence reduces the number of generated test cases compared to the baseline, which tries to generate test cases whenever a new branch is covered. Furthermore, the results depicted in Table 6.4 reflect average values over multiple experiments. It was observed that the generated context and the order and choice of summaries during the applicability check influence the achieved performance.

Comparison of different Heuristics The baseline is put to the test in comparison with a non-merged-based (NMB) and a merge-based (MB) concolic approach using summarization in Table 6.5. As profiling has shown that a lot of time is spent on solving the symbolic expressions during the applicability checking, two different heuristics are evaluated. Table 6.5 shows the results of the comparison between the baseline, the NMB, and MB concolic approaches using summarization. The timeout is set to 10 minutes, and the cycle bound is set to 25 cycles. The cumulative application checking time for the summaries is contained in the total runtimes (**t[s]**) for the respective approaches.

While the algorithm for finding applicable summaries executes very fast for the NMB approach due to the use of concrete valuations, the explicit enumeration of all feasible paths through one cycle degenerates quickly [Gro & Völ+ 22a]. For deeper nested execution contexts, this way of exploring the CFG of a PLC program is intractable due to the path explosion problem emerging from the cyclic behavior.

Considering the compromise of still executing concolically while merging at join points, additional symbolic checking has to be undertaken as information is partially lost [Gro & Völ+ 22a]. This limitation stems from the way how the concrete stores of the execution contexts are currently merged, resulting in an under-approximation and hence loss of context information during the summary application [Gro & Völ+ 22a]. The increased running times can be explained by the engine's effort to alleviate this context loss. Therefore, for each concretely non-applicable summary, a symbolic check has to be performed additionally, offloading expensive queries to the SMT solver [Gro & Völ+ 22a].

Reusing Summaries across Program Versions For the sake of completeness, the results for reusing summaries across program versions are discussed next despite the poor results for the summary application check. Table 6.6 shows the results of the validity checking algorithm for the generated summaries of the PLCopen

Table 6.5: Comparison of the baseline, the non-merge-based concolic and the merge-based technique on the PLCopen Safety suite. The rows highlighted in blue show results where the other techniques outperformed the baseline CSE.

Table adapted from Table II in [Gro & Völ+ 22a].

Function Block / Program	CSE w/o SA			NMB + Sum.			MB + Sum.		
	cov. [%]	**T [#]**	**t[s]**	**cov. [%]**	**T [#]**	**t[s]**	**cov. [%]**	**T [#]**	**t[s]**
`Antivalent`	80.26	21	10.47	80.26	14	TO	80.26	21	14.66
`EDM`	97.86	64	10.99	97.86	39	TO	97.86	79	138.35
`Emergency_Stop`	94.29	26	9.75	94.29	15	TO	94.29	24	**9.1**
`Enable_Switch`	93.42	31	12.97	93.42	23	TO	93.42	33	14.37
`Equivalent`	93.75	24	8.95	93.75	16	TO	93.75	26	13.93
`ESPE`	94.25	25	10.14	94.25	15	TO	94.25	24	**9.19**
`Guard_Locking`	97.56	38	16.60	97.56	27	TO	97.56	47	27.88
`Guard_Monitoring`	93.02	34	19.07	93.02	23	TO	93.02	38	25.65
`Mode_Selector`	94.44	29	14.67	90.28	40	TO	95.83	62	187.21
`Muting_Seq`	95.90	54	56.66	86.89	35	TO	95.90	68	111.10
`Out_Control`	95.71	31	11.64	95.71	22	TO	95.71	37	16.97
`Safe_Stop`	94.88	31	11.64	94.88	20	TO	94.88	40	220.57
`Limit_Speed`	97.87	41	24.38	80.85	22	TO	97.87	54	376.83
`Safety_Request`	95.56	36	15.99	95.56	21	TO	95.56	36	24.14
`Safety_Sensor`	91.67	69	14.49	69.87	28	TO	91.67	85	158.43
`TH_Control_T_II`	96.51	38	18.33	96.51	22	TO	96.51	40	23.10
`TH_Control_T_III`	96.36	46	33.34	96.36	22	TO	96.36	48	57.34

Safety suite after reconfigurations have been performed to the respective FBs. The

Table 6.6: Runtimes for validity checking of arbitrary changes to the PLCopen Safety suite.
Table adapted from Table III in [Gro & Völ+ 22a].

Function Block / Program	S [#]	Phase 1 [s]	Phase 2 [s]	Phase 3 [s]	Valid [#]	Invalid [#]
Antivalent	36	0	0.30	0.32	36	0
EDM	65	0	1.42	0.02	64	1
Emergency_Stop	25	0	0.10	0.03	24	1
Enable_Switch	32	0	0.13	0.04	32	0
Equivalent	34	0	0.14	0.03	33	1
ESPE	25	0	0.10	0.06	23	2
Guard_Locking	36	0	0.23	0.02	33	3
Guard_Monitoring	40	0	0.22	0.03	38	2
Mode_Selector	59	0	0.64	0.09	58	1
Muting_Seq	56	0	1.18	0.60	53	3
Out_Control	31	0	0.12	0.08	30	1
Safe_Stop	93	0	0.75	0.03	89	4
Limit_Speed	121	0	0.84	0.02	117	4
Safety_Request	36	0	0.22	0.05	35	1
Safety_Sensor	67	0	1.93	0.03	66	1
TH_Control_T_II	39	0	0.17	0.04	38	1
TH_Control_T_III	53	0	0.36	0.02	50	3

reconfigurations were performed manually and had no semantic reasoning behind them. This resulted in only one or two reconfigurations per FB as it required a manual check for correctness. In general, the results of the second and third phase are dependent on the encoding of the problem instance and the performance of the used SMT solver. The algorithm performs as it was expected from the results obtained in the respective source [God & Lah+ 11] and underlines that the bottleneck in efficiently reusing summaries across program versions remains in their application checking during SE.

6.3 Test Suite Augmentation

Table 6.7 shows the results of the TSA algorithm applied to selected reconfiguration scenarios of the PPU benchmark. The first column denotes the analyzed reconfiguration scenario (cf. Figure 6.2). The second column relates the number of untouched change-annotated vertices (b_u **[#]**) to the number of total change-annotated vertices (b_{ca} **[#]**) in the program using the test suite of the program before the reconfiguration. This ratio gives an estimate of how well the previous test suite is capable of finding divergences using SSE.

Table 6.7: Results of the TSA using Algorithm 7 for selected reconfiguration scenarios of the PPU.
Table adapted from Table 3. in [Gro & Völ+ 22b].

PPU Evolution	b_u **[#]** / b_{ca}**[#]**	T_{ca}**[#]**	FDC		BSE		T_{diff}**[#]**
			Q_{div} [#]	**t [s]**	T_{div} [#]	**t [s]**	
Scenario_{0 → 1}	0/0	0	0	0	0	0	0
Scenario_{0 → 2}	1/12	45	2	1.77	52	54.99	23
Scenario_{2 → 3}	21/50	55	21	19.49	1269	3423.94	1269

The third column denotes the number of test cases (T_{ca}[#]) of the test suite that exercise any number of change-annotated vertices b_{ca} in the change-annotated PLC program [Gro & Völ+ 22b]. The generated test cases are succinct with regard to the required number of cycles to reach a particular specific coverage metric. Therefore test cases that cover deeper nested code fragments can share a partial prefix with other test cases [Gro & Völ+ 22b]. As SSE requires a seed in the form of a test case as an input, this leads to a natural limitation of the SSE approach. An increased analysis time for the subsequent phases is expected for programs that exhibit cyclic behavior, such as in the PLC domain.

The fourth column represents the analysis results of Algorithm 8 in Section 5.2.3 for finding divergent execution contexts. It lists the number of derived divergent contexts (Q_{div}[#]) and the total time it took to perform the analysis (**t [s]**) for each representative test case derived from the prior phase.

The fifth column represents the analysis results of propagating the divergent execution contexts by initiating a BSE for each divergent context found. It is divided into the number of divergent test cases (T_{div}[#]) generated by the BSE using the corresponding triggering test cases as a seed for the concolic execution and the total runtime (**t [s]**).

Last but not least, the sixth column denotes the number of difference-revealing test cases (T_{diff}[#]) that were derived from the divergent test cases T_{div}[#] by checking the observable behavior of the old and the new version of the program.

Discussion The first row of Table 6.7 shows that the TSA algorithm did not perform any analysis as Scenario_{0 → 1} only introduced a mechanical reconfiguration (see Table 6.1 and Figure 6.2). Scenario_{0 → 2} consists of 12 change-annotated vertices, of which one is not traversed by the test suite T generated for Scenario_0. This shows that the old test suite T is not adequate enough to test Scenario_2 with regard to the respective coverage measure. In order to guarantee an exhaustive analysis, a directed TSG must be performed to derive a test case traversing the untraversed change-annotated vertex. This is disregarded in the subsequent analysis, and all 45 test cases from the prior test suite are selected as T_{ca}[#]. For each test case in T_{ca}[#], Algorithm 8 of Chapter 5 is executed, yielding

two divergent contexts in under two seconds. For each divergent context, a BSE is started from the point of divergence resulting in 52 divergent test inputs in under a minute. Of all those divergent test inputs, only 23 are truly difference-revealing and need to be subsequently analyzed by the developer to distinguish between expected divergences and regression bugs. As the increase in **LOCs** depicted in Table 6.3 already suggests, `Scenario_`$\{2 \rightarrow 3\}$ has even more untraversed change-annotated vertices, divergent execution contexts, and difference-revealing test cases. While the generation time of roughly an hour is an acceptable time, the amount of generated difference-revealing test cases that have to manually be analyzed by the developer is not. The reason for the amount of generated test cases during BSE for `Scenario_`$\{2 \rightarrow 3\}$ is explained with the help of Figure 6.4. The

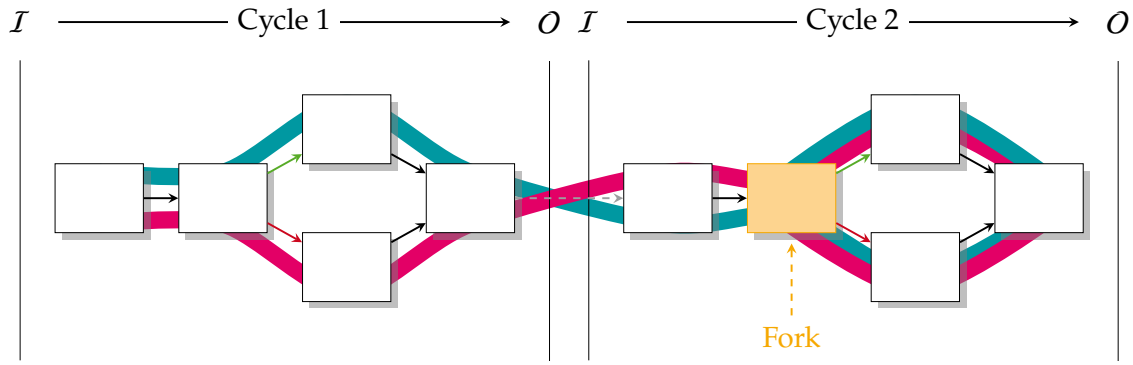


Figure 6.4: Paths taken by the generated test cases under divergent test inputs during BSE.

divergent execution contexts and the respective divergent test inputs represent the contexts that triggered either a diverging concrete execution or were generated because of potential possible divergences at the four-way fork. Assume the blue and the red path denote divergent test inputs derived in the analysis prior to BSE. In the example, the branch condition within the block denoted by orange is only dependent on the input valuations. This, therefore, leads to a fork and the generation of two test cases for the true and the false branch inheriting the prefix of the first cycle. The same is done for the other divergent test input denoted by the red path, and again two test cases covering both branches are derived. In total, BSE generates four test cases while the initial two divergent test inputs would be sufficient to yield the same coverage measure in case of branch coverage.

7 Conclusion

The evolution of PLC software involves the addition, modification, and removal of functionality. In order to guarantee that a PLC program is adequately tested after such a reconfiguration with regard to a specified coverage metric, the test suite must also evolve.

The goal of this thesis was to provide a "push-button" analysis for the generation of test cases after a reconfiguration. This tool should be usable during static reconfiguration, i.e., when the entire system is stopped during maintenance, or during virtual commissioning. In these scenarios, the test input data can be fed and supervised to observe the impact of the reconfiguration on the CPPS by the developer. To achieve this goal, this thesis investigated two complementing approaches. The general rationale was to improve efficiency by not doing redundant work when trying to generate test cases for a PLC program and augmenting the test suite for the new PLC program after a reconfiguration.

Test Suite Generation Automatic TSG is a well-established technique used to generate test suites adhering to structural coverage metrics of PLC software [Sim & Fri+ 15; Boh & Sim+ 16; Gro & Völ+ 22a]. However, in the face of reconfigurations, a new analysis would not consider the effort of the prior analysis. In order to reduce this redundancy in TSG after a structural reconfiguration to the PLC software has occurred, symbolic summaries of specific parts of the program have been derived, cached, and reused to investigate whether they can benefit the subsequent analysis. As code untouched from reconfigurations will result in equivalent path conditions, the goal was to determine whether the application of summaries can aid in increasing the analysis' performance. For this purpose, a combination of state-of-the-art CSE and SA algorithms for TSG and summary reuse [Can & God 19] have been implemented and evaluated. The evaluation of the prototypical evaluation on the PLCOpen Safety suite in Section 6.2 clearly showed the ineffectiveness of using summarization of FBs during TSG for reconfigurable PLC software.

Test Suite Augmentation After a reconfiguration, the generated test suite via automatic TSG might not be adequate enough anymore with regard to the coverage metric to ensure the absence of regressions in the new program version. An indispensable part of RT poses TSA, which guides the TSG toward the reconfigured behavior and increases the chances of deriving difference-revealing test cases which expose behavioral differences between the program and its reconfigured version.

Intuitively, two things are needed to guarantee that a test suite is adequate enough with regard to some coverage metric to test the behavior after a reconfiguration: (1) test cases must reach potentially affected areas, and (2) test cases must account for the state of the software and the effects of changes. Reaching potentially affected areas (1) must occur along different, relevant paths following specific data- and control dependency chains. In Chapter 3, it was shown that a static CIA degenerates for the analysis of changes after a reconfiguration to the PLC software. It is often imprecise and misses important pruning and prioritization opportunities. The execution of modified instructions does not mean that they are necessarily difference-revealing because the subdomains do not need to be homogenous with regard to the failure [Wey & Jen 91]. Semantic change-impact analysis may optimize the search for relevant areas during SE and prune unrelated paths, but it was out of the scope for this thesis.

This thesis presented the implementation of state-of-the-art CSE algorithms for TSA in the area of PLC programs. Test generation is guided toward the changed behavior using a technique known as four-way forking [Pal & Kuc+ 16; Kuc & Pal+ 18] from SSE. The old and new PLC software are executed in the same CSE instance through the use of a CAP to account for the effects of the reconfiguration and increase the chances of generating difference-revealing test cases. The prototypical implementation was evaluated using domain-specific benchmarks such as the PLCopen Safety library and the PPU. It exposed the limitations in applicability and effectiveness of the used techniques for safeguarding PLC software subject to frequent reconfigurations as found in CPPSs.

While the generation of difference-revealing test cases (2) is a good step toward increasing the confidence in the absence of errors, partition testing might often not be better than random input generation. Hence exhaustive four-way forking is necessary to guarantee the absence of errors. Exhaustivenes, however, comes with a price rendering it often intractable [Nol & Ngu+ 19]. Another possibility of improving the effectiveness of SE to find regression errors and faults is achieved by using the concept of fault-based testing. Typically, this is achieved by instrumenting the CFG of the reconfigured program with a series of *enhanced cross-version checks* [Kuc & Pal+ 18; Jam & Fra+ 13].

7.1 Outlook

Several issues remain for the implemented TSG. Currently, the summarization supports only whole FBs without nested callees. Future work should investigate arbitrary code summarization and also allow multiple, nested FBs during summarization. In general, I assume that the benefits of summarization would be more visible when analyzing more extensive and complex programs than the PLCopen Safety suite. Furthermore, incremental solving should be investigated in more depth. While it was prototypically implemented for summary pruning to combat the lack of context information during the summary application, the results were

poor.

The amount of generated difference-revealing test cases by the TSA is not manually analyzable. Potential solutions such as test case prioritization or other test suite reduction techniques from RT should be evaluated in order to narrow down the required test cases, which must be checked by the developer after a reconfiguration to uncover potential regressions.

Figure 7.1 summarizes state-of-the-art techniques and puts them in comparison concerning their encoding. A significant drawback of using symbolic summaries

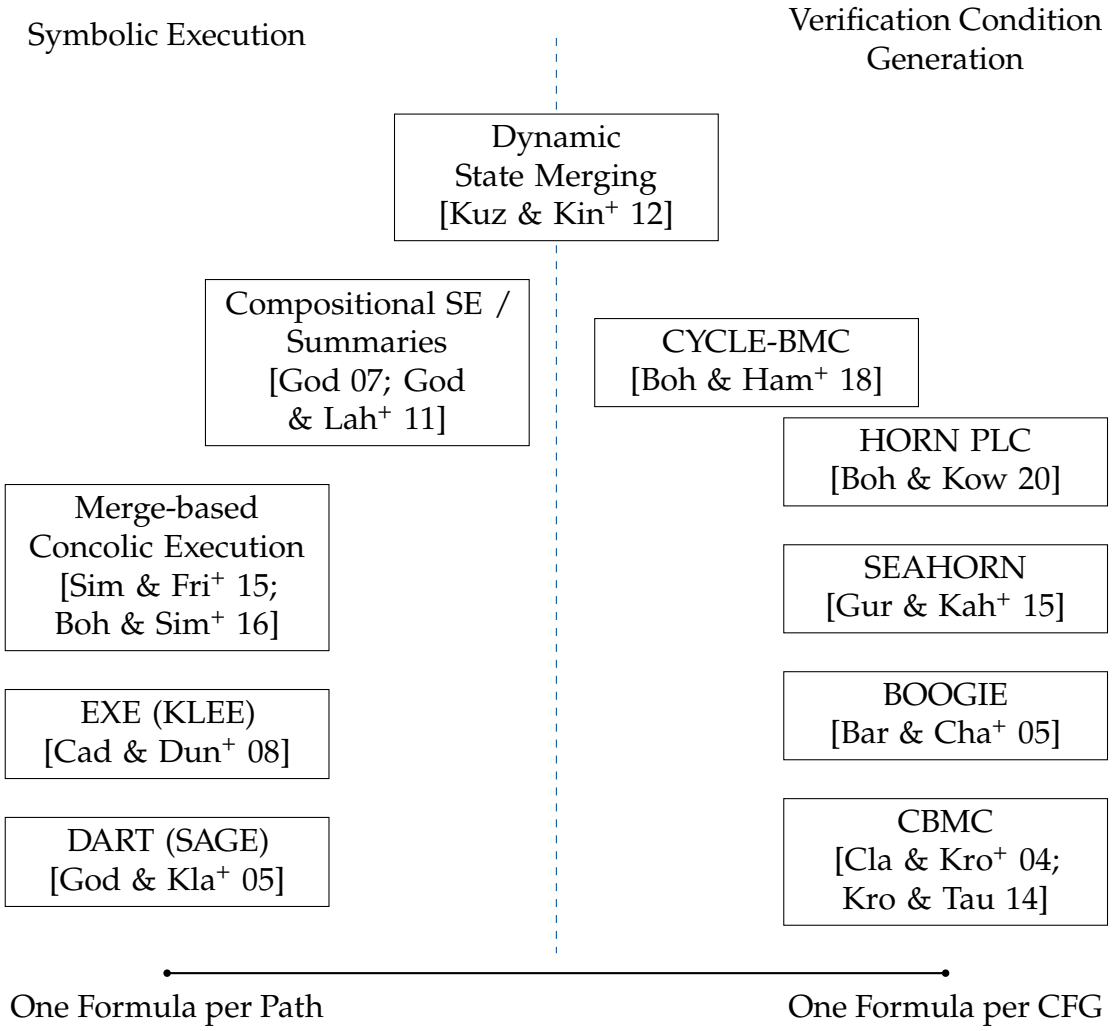


Figure 7.1: Relationship between SE and VCG.
Figure adapted from [Kuz & Kin+ 14; Can & God 19].

and state merging in SE is the increased solving cost. This thesis investigated dynamically executing the CFG and merging execution states at control flow join points, where all feasible subpaths have been explored before the merge. This is

quite similar to static state merging from VCG, with the exception that this thesis under-approximates the execution paths through the CFG.

Furthermore, the IR under-approximates the behavior of the analyzed CFG by using function cloning to avoid invalid interprocedural paths. Over-approximation boosts merging and analysis speed at the cost of introducing spurious behavior. An interesting research question would be to investigate the trade-off between the progress gained through spuriousness and the time lost refining the counterexamples to derive test cases. Experiments with VCG for the verification of PLC programs [Boh & Ham+ 18; Boh & Kow 20; Boh 21] have shown that this technique outperforms other state-of-the-art verification techniques for PLC programs. Especially interesting is the fact that during the summary generation in Chapter 4, the implementation of SE is based on the concepts of [Boh & Ham+ 18]. In future work, experiments adopting VCG and intermediate merging strategies may move Cycle-Bmc more into the direction of SE and pose a competitive solution for TSG with path-based summary reuse.

Operational Semantics

A

The value of an expression depends on the respective store. The concrete and symbolic operational semantics of expressions are defined over configurations $\langle e, \rho \rangle$ and $\langle e, \sigma \rangle$ consist of the expression $e \in \boldsymbol{E}$ and the concrete or symbolic store, ρ and σ, respectively. The big-step operational semantics of expressions are specified in terms of the transition relation operator "$\rightarrow$" as follows: $\langle e, \rho \rangle \rightarrow d$ for the evaluation of the concrete configuration and $\langle e, \sigma \rangle \rightarrow \gamma$ for the evaluation of the symbolic configuration in their respective domains [Nie & Nie$^+$ 99]. Hence, for example, $\langle e := x > 0, \rho := \{x \mapsto 1\} \rangle \rightarrow d := \mathit{true}$ means that expression $x > 0$ evaluates to *true* using the concrete valuation $x \mapsto 1$.

The symbolic operational semantics of expressions are defined in terms of Z3 expressions and their internal semantics [dMou & Bjø 08]. This facilitates the entire implementation of a type system and gives rise to the powerful way Z3 handles their expressions in memory.

The operational semantics of the expressions and instructions are used throughout this thesis in Chapter 4 and Chapter 5 within the SE algorithms.

Definition A.1: Concrete Operational Semantics of Expressions

$$\frac{}{\langle i, \rho \rangle \rightarrow i} \; i \in \mathbb{I} \qquad \frac{}{\langle v_a, \rho \rangle \rightarrow \rho(v_a)} \; v_a \in \boldsymbol{V}_A$$

NEGATION
$$\frac{\langle a, \rho \rangle \rightarrow i}{\langle -a, \rho \rangle \rightarrow -i}$$

EXPONENTIATION
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 ** a_2, \rho \rangle \rightarrow i_1 ** i_2}$$

MULTIPLICATION
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 * a_2, \rho \rangle \rightarrow i_1 * i_2}$$

DIVISION
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 / a_2, \rho \rangle \rightarrow i_1 / i_2}$$

MODULO
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 \text{ MOD } a_2, \rho \rangle \rightarrow i_1 \text{ MOD } i_2}$$

ADDITION
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 + a_2, \rho \rangle \rightarrow i_1 + i_2}$$

SUBTRACTION
$$\frac{\langle a_1, \rho \rangle \rightarrow i_1 \qquad \langle a_2, \rho \rangle \rightarrow i_2}{\langle a_1 - a_2, \rho \rangle \rightarrow i_1 - i_2}$$

Definition A.2: Concrete Operational Semantics of Boolean Expressions

$$\frac{}{\langle true, \rho\rangle \rightarrow true}\, true \in \mathbb{B} \qquad \frac{}{\langle false, \rho\rangle \rightarrow false}\, false \in \mathbb{B}$$

$$\frac{}{\langle v_b, \rho\rangle \rightarrow \rho(v_b)}\, v_b \in \boldsymbol{V}_B \qquad \textsc{Complement}\ \frac{\langle b, \rho\rangle \rightarrow t}{\langle \neg b, \rho\rangle \rightarrow \neg t}$$

$$\textsc{Less Than}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 < a_2, \rho\rangle \rightarrow i_1 < i_2} \qquad \textsc{Greater Than}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 > a_2, \rho\rangle \rightarrow i_1 > i_2}$$

$$\textsc{Less Or Equal Than}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 \leq a_2, \rho\rangle \rightarrow i_1 \leq i_2} \qquad \textsc{Greater Or Equal Than}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 \geq a_2, \rho\rangle \rightarrow i_1 \geq i_2}$$

$$\textsc{Equality}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 = a_2, \rho\rangle \rightarrow i_1 = i_2} \qquad \textsc{Inequality}\ \frac{\langle a_1, \rho\rangle \rightarrow i_1 \qquad \langle a_2, \rho\rangle \rightarrow i_2}{\langle a_1 <> a_2, \rho\rangle \rightarrow i_1 \neq i_2}$$

$$\textsc{Boolean And}\ \frac{\langle b_1, \rho\rangle \rightarrow t_1 \qquad \langle b_2, \rho\rangle \rightarrow t_2}{\langle b_1 \text{ AND } b_2, \rho\rangle \rightarrow t_1 \wedge t_2} \qquad \textsc{Boolean Xor}\ \frac{\langle b_1, \rho\rangle \rightarrow t_1 \qquad \langle b_2, \rho\rangle \rightarrow t_2}{\langle b_1 \text{ XOR } b_2, \rho\rangle \rightarrow t_1 \veebar t_2}$$

$$\textsc{Boolean Or}\ \frac{\langle b_1, \rho\rangle \rightarrow t_1 \qquad \langle b_2, \rho\rangle \rightarrow t_2}{\langle b_1 \text{ OR } b_2, \rho\rangle \rightarrow t_1 \vee t_2}$$

Definition A.3: Symbolic Operational Semantics of Expressions

$$\frac{}{\langle v_a, \sigma\rangle \to \sigma(v_a)}\, v_a \in \boldsymbol{V}_A$$

Negation
$$\frac{\langle a, \sigma\rangle \to \gamma}{\langle -a, \sigma\rangle \to -(\gamma)}$$

Exponentiation
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 ** a_2, \sigma\rangle \to **(\gamma_1, \gamma_2)}$$

Multiplication
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 * a_2, \sigma\rangle \to *(\gamma_1, \gamma_2)}$$

Division
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 / a_2, \sigma\rangle \to /(\gamma_1, \gamma_2)}$$

Modulo
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 \text{ MOD } a_2, \sigma\rangle \to \text{ MOD } (\gamma_1, \gamma_2)}$$

Addition
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 + a_2, \sigma\rangle \to +(\gamma_1, \gamma_2)}$$

Subtraction
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 - a_2, \sigma\rangle \to -(\gamma_1, \gamma_2)}$$

$$\frac{}{\langle v_b, \sigma\rangle \to \sigma(v_b)}\, v_b \in \boldsymbol{V}_B$$

Complement
$$\frac{\langle b, \sigma\rangle \to \gamma}{\langle \neg b, \sigma\rangle \to \neg(\gamma)}$$

Less Than
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 < a_2, \sigma\rangle \to <(\gamma_1, \gamma_2)}$$

Greater Than
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 > a_2, \sigma\rangle \to >(\gamma_1, \gamma_2)}$$

Less Or Equal Than
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 \leq a_2, \sigma\rangle \to \leq(\gamma_1, \gamma_2)}$$

Greater Or Equal Than
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 \geq a_2, \sigma\rangle \to \geq(\gamma_1, \gamma_2)}$$

Equality
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 = a_2, \sigma\rangle \to =(\gamma_1, \gamma_2)}$$

Inequality
$$\frac{\langle a_1, \sigma\rangle \to \gamma_1 \qquad \langle a_2, \sigma\rangle \to \gamma_2}{\langle a_1 <> a_2, \sigma\rangle \to \neq(\gamma_1, \gamma_2)}$$

Boolean And
$$\frac{\langle b_1, \sigma\rangle \to \gamma_1 \qquad \langle b_2, \sigma\rangle \to \gamma_2}{\langle b_1 \text{ AND } b_2, \sigma\rangle \to \wedge(\gamma_1, \gamma_2)}$$

Boolean Xor
$$\frac{\langle b_1, \sigma\rangle \to \gamma_1 \qquad \langle b_2, \sigma\rangle \to \gamma_2}{\langle b_1 \text{ XOR } b_2, \sigma\rangle \to \veebar(\gamma_1, \gamma_2)}$$

Boolean Or
$$\frac{\langle b_1, \sigma\rangle \to \gamma_1 \qquad \langle b_2, \sigma\rangle \to \gamma_2}{\langle b_1 \text{ OR } b_2, \sigma\rangle \to \vee(\gamma_1, \gamma_2)}$$

The big-step operational semantics of instructions are defined over configurations $\langle I, s \rangle$. The judgment has the form $\langle I, s \rangle \to s'$, where s' is the succeeding execution state after executing the instruction I in execution state s.

Definition A.4: Concrete Operational Semantics of Instructions

GOTO

$$\frac{}{\langle \textbf{goto}\ b_{\ell'}, b_\ell, \rho, \sigma, \pi \rangle \to b_{\ell'}, \rho, \sigma, \pi}$$

SEQUENCE

$$\frac{\langle I_1, b_\ell, \rho, \sigma, \pi \rangle \to b_{\ell''}, \rho'', \sigma'', \pi'' \qquad \langle I_2, b_{\ell''}, \rho'', \sigma'', \pi'' \rangle \to b_{\ell'}, \rho', \sigma', \pi'}{\langle \textbf{sequence}(I_1, I_2), b_\ell, \rho, \sigma, \pi \rangle \to b_{\ell'}, \rho', \sigma', \pi'}$$

ASSIGNMENT

$$\frac{\langle e, \rho \rangle \to d}{\langle \textbf{assign}(v, e), b_\ell, \rho, \sigma, \pi \rangle \to b_\ell, \rho[v \mapsto d], \sigma, \pi}$$

IF – GOTO TRUE

$$\frac{\langle b, \rho \rangle \to \text{true} \qquad \langle \textbf{goto}\ b_{\ell'}, \rho, \sigma, \pi \rangle \to b_{\ell'}, \rho', \sigma', \pi'}{\langle \textbf{ite}(b, \textbf{goto}\ b_{\ell'}, \textbf{goto}\ b_{\ell''}), \rho, \sigma, \pi \rangle \to b_{\ell'}, \rho', \sigma', \pi'}$$

IF – GOTO FALSE

$$\frac{\langle b, \rho \rangle \to \text{false} \qquad \langle \textbf{goto}\ b_{\ell''}, \rho, \sigma, \pi \rangle \to b_{\ell''}, \rho'', \sigma'', \pi''}{\langle \textbf{ite}(b, \textbf{goto}\ b_{\ell'}, \textbf{goto}\ b_{\ell''}), \rho, \sigma, \pi \rangle \to b_{\ell''}, \rho'', \sigma'', \pi''}$$

CALL

$$\frac{\begin{array}{c}\langle e_1, \rho \rangle \to d_1, \ldots, \langle e_m, \rho \rangle \to d_m \\ \langle b_\ell, \rho[e_1 \mapsto d_1, \ldots, e_m \mapsto d_m], \sigma, \pi \rangle \to^* b_{\ell'}, \rho', \sigma', \pi'\end{array}}{\begin{array}{c}\langle v_1, \ldots, v_n \coloneqq \textbf{call}\ G(e_1, \ldots, e_m), b_\ell, \rho, \sigma, \pi \rangle \\ \to b_{\ell'}, \rho[v_1 \mapsto \rho'(v_1), \ldots, \rho'(v_n)], \sigma', \pi'\end{array}}$$

Definition A.5: Symbolic Operational Semantics of Instructions

$$\textsc{Assignment}\quad \frac{\langle e, \rho\rangle \to d}{\langle \textbf{assign}(v, e), b_\ell, \rho, \sigma, \pi\rangle \to b_\ell, \rho[v \mapsto d], \sigma, \pi}$$

$$\textsc{If - Goto True}\quad \frac{\langle b, \sigma\rangle \to \beta \qquad \pi \wedge \beta \text{ is SAT} \qquad \langle \textbf{goto } b_{\ell'}, \rho, \sigma, \pi \wedge \beta\rangle \to b_{\ell'}, \rho', \sigma', \pi'}{\langle \textbf{ite}(b, \textbf{goto } b_{\ell'}, \textbf{goto } b_{\ell''}), \rho, \sigma, \pi\rangle \to b_{\ell'}, \rho', \sigma', \pi'}$$

$$\textsc{If - Goto False}\quad \frac{\langle b, \rho\rangle \to \beta \qquad \pi \wedge \neg\beta \text{ is SAT} \qquad \langle \textbf{goto } b_{\ell''}, \rho, \sigma, \pi \wedge \neg\beta\rangle \to b_{\ell''}, \rho'', \sigma'', \pi''}{\langle \textbf{ite}(b, \textbf{goto } b_{\ell'}, \textbf{goto } b_{\ell''}), \rho, \sigma, \pi\rangle \to b_{\ell''}, \rho'', \sigma'', \pi''}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}\langle e_1, \sigma\rangle \to \gamma_1, \ldots, \langle e_m, \sigma\rangle \to \gamma_m \\ \langle b_\ell, \rho, \sigma[e_1 \mapsto \gamma_1, \ldots, e_m \mapsto \gamma_m], \pi\rangle \to^* b_{\ell'}, \rho', \sigma', \pi'\end{array}}{\begin{array}{c}\langle v_1, \ldots, v_n \coloneqq \textbf{call } G(e_1, \ldots, e_m), b_\ell, \rho, \sigma, \pi\rangle \\ \to b_{\ell'}, \rho', \sigma[v_1 \mapsto \sigma'(v_1), \ldots, \sigma'(v_n)], \pi'\end{array}}$$

Last but not least, the operational semantic of a program is presented.

Definition A.6: Operational Semantic of a Program

$$\textsc{Cycle}\quad \frac{\begin{array}{c}\forall v \in \boldsymbol{V} \setminus \boldsymbol{V}_{input}.\rho''[v \mapsto \rho(v)] \\ \forall v \in \boldsymbol{V}_{input}.\rho''[v \mapsto d] \qquad \langle b_\ell, \rho'', \sigma'', \pi''\rangle \to^* b_{\ell'}, \rho', \sigma', \pi'\end{array}}{\langle \textbf{cycle}, b_\ell, \rho, \sigma, \pi\rangle \to b_{\ell'}, \rho', \sigma', \pi'}$$

While there exists no explicit instruction in the IR to capture the semantics of the PLC cycle, it is implicitly encoded by an edge in the respective CFG of the main program.

List of Figures

List of Tables

List of Definitions

List of Examples

List of Acronyms

GTT generalized test tables

RTT relational test table

AST abstract syntax tree

BB basic block

BMC bounded model checking

BSE bounded symbolic execution

CAP change-annotated program

CE concolic execution

CERN European Organization for Nuclear Research

CFA control-flow automaton

CFG control-flow graph

CG call graph

CIA change impact analysis

CPPS cyber-physical production system

CSE compositional symbolic execution

CTL Computation Tree Logic

DFS depth-first search

DiSE directed incremental symbolic execution

DS Digital Shadow

DSE differential symbolic execution

FB function block

FBD Function Block Diagram

HIL hardware-in-the-loop

ICT information and communications technologies

iDiSE interprocedural directed incremental symbolic execution

IIoT Industrial Internet of Things

IL Instruction List

IoP Internet of Production

IR intermediate representation

LBE large-block encoding

LD Ladder Diagram

MAS multi-agent system

OPC UA OPC Unified Architecture

PAF program analysis framework

PLC programmable logic controller

POU program organization unit

PPU Pick and Place Unit

PRV partition-based regression verification

RT regression testing

RVT Regression Verification Tool

SA static analysis

SE symbolic execution

SFC Sequential Function Chart

SMT satisfiability modulo theories

SOA service-oriented architecture

SSA static single assignment

SSE shadow symbolic execution

ST Structured Text

SUT system under test

TAC three-address code

TSA test suite augmentation

TSG test suite generation

TSM test suite maintenance

VC verification condition

VCG verification condition generation

VSA value set analysis

Bibliography

[Adi & Dar+ 14] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, V. M. G. Suárez, and J. O. Blech, "Modelling and Formal Verification of Timing Aspects in Large PLC Programs", *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3333–3339, 2014, 19th IFAC World Congress, ISSN: 1474-6670. DOI: https://doi.org/10.3182/20140824-6-ZA-1003.01279. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016421208.

[Adi & Dar+ 15] B. F. Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying Model Checking to Industrial-Sized PLC Programs", *IEEE Trans. Ind. Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015. DOI: 10.1109/TII.2015.2489184. [Online]. Available: https://doi.org/10.1109/TII.2015.2489184.

[Aho & Set+ 86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley series in computer science / World student series edition). Addison-Wesley, 1986, ISBN: 0-201-10088-6. [Online]. Available: https://www.worldcat.org/oclc/12285707.

[All 70] F. E. Allen, "Control flow analysis", in *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, R. S. Northcote, Ed., ACM, 1970, pp. 1–19. DOI: 10.1145/800028.808479. [Online]. Available: https://doi.org/10.1145/800028.808479.

[Ana & God+ 08] S. Anand, P. Godefroid, and N. Tillmann, "Demand-Driven Compositional Symbolic Execution", in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 367–381. DOI: 10.1007/978-3-540-78800-3_28. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3%5C_28.

[Avg & Reb+ 14] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing Symbolic Execution with Veritesting", in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds., ACM, 2014, pp. 1083–1094. DOI: 10.1145/2568225.2568293. [Online]. Available: https://doi.org/10.1145/2568225.2568293.

[Bac & Per+ 13] J. D. Backes, S. Person, N. Rungta, and O. Tkachuk, "Regression Verification Using Impact Summaries", in *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, E. Bartocci and C. R. Ramakrishnan, Eds., ser. Lecture Notes in Computer Science, vol. 7976, Springer, 2013, pp. 99–116. DOI: 10.1007/978-3-642-39176-7_7. [Online]. Available: https://doi.org/10.1007/978-3-642-39176-7%5C_7.

[Bai & Kat 08] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008, ISBN: 978-0-262-02649-9.

[Bal & Cop+ 18] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques", *ACM Comput. Surv.*, vol. 51, no. 3, 50:1–50:39, 2018. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[Bar & Cha+ 05] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs", in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., ser. Lecture Notes in Computer Science, vol. 4111, Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17. [Online]. Available: https://doi.org/10.1007/11804192%5C_17.

[Bec & Bib+ 21] F. Becker, P. Bibow, M. Dalibor, A. Gannouni, V. Hahn, C. Hopmann, M. Jarke, I. Koren, M. Kröger, J. Lipp, J. Maibaum, J. Michael, B. Rumpe, P. Sapel, N. Schäfer, G. J. Schmitz, G. Schuh, and A. Wortmann, "A Conceptual Model for Digital Shadows in Industry and Its Application", in *Conceptual Modeling - 40th International Conference, ER 2021, Virtual Event, October 18-21, 2021, Proceedings*, A. K. Ghose, J. Horkoff, V. E. S. Souza, J. Parsons, and J. Evermann, Eds., ser. Lecture Notes in Computer Science, vol. 13011, Springer, 2021, pp. 271–281. DOI: 10.1007/978-3-030-89022-3_22. [Online]. Available: https://doi.org/10.1007/978-3-030-89022-3%5C_22.

[Bec & Cha+ 17] B. Beckert, S. Cha, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Generalised Test Tables: A Practical Specification Language for Reactive Systems", in *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*, N. Polikarpova and S. A. Schneider, Eds., ser. Lecture Notes in Computer Science, vol. 10510, Springer, 2017, pp. 129–144. DOI: 10.1007/978-3-319-66845-1_9. [Online]. Available: https://doi.org/10.1007/978-3-319-66845-1%5C_9.

[Bec & Mun+ 19] B. Beckert, J. Mund, M. Ulbrich, and A. Weigl, "Formal Verification of Evolutionary Changes", in *Managed Software Evolution*, R. H. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin, Eds., Springer, 2019, pp. 309–332. DOI: 10.1007/978-3-030-13499-0_11. [Online]. Available: https://doi.org/10.1007/978-3-030-13499-0%5C_11.

[Bec & Ulb+ 15] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Regression Verification for Programmable Logic Controller Software", in *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, M. J. Butler, S. Conchon, and F. Zaïdi, Eds., ser. Lecture Notes in Computer Science, vol. 9407, Springer, 2015, pp. 234–251. DOI: 10.1007/978-3-319-25423-4_15. [Online]. Available: https://doi.org/10.1007/978-3-319-25423-4%5C_15.

[Bey & Lem 16] D. Beyer and T. Lemberger, "Symbolic Execution with CEGAR", in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 9952, 2016, pp. 195–211. DOI: 10.1007/978-3-319-47166-2_14. [Online]. Available: https://doi.org/10.1007/978-3-319-47166-2%5C_14.

[Bia & Bra+ 12] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.PLC: a verification platform for programmable logic controllers", in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, M. Goedicke, T. Menzies, and M. Saeki, Eds., ACM, 2012, pp. 338–341. DOI: 10.1145/2351676.2351741. [Online]. Available: https://doi.org/10.1145/2351676.2351741.

[Bia 16] S. Biallas, "Verification of Programmable Logic Controller Code using Model Checking and Static Analysis", Ph.D. dissertation, RWTH Aachen University, Germany, 2016, ISBN: 978-3-8440-

4711-0. [Online]. Available: http://publications.rwth-aachen.de/record/668156.

[Bib & Dal+ 20] P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, and A. Wortmann, "Model-Driven Development of a Digital Twin for Injection Molding", in *Advanced Information Systems Engineering - 32nd International Conference, CAiSE 2020, Grenoble, France, June 8-12, 2020, Proceedings*, S. Dustdar, E. Yu, C. Salinesi, D. Rieu, and V. Pant, Eds., ser. Lecture Notes in Computer Science, vol. 12127, Springer, 2020, pp. 85–100. DOI: 10.1007/978-3-030-49435-3_6. [Online]. Available: https://doi.org/10.1007/978-3-030-49435-3%5C_6.

[Böh & dS O+ 13] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-Based Regression Verification", in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., IEEE Computer Society, 2013, pp. 302–311. DOI: 10.1109/ICSE.2013.6606576. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606576.

[Boh & Ham+ 18] D. Bohlender, D. Hamm, and S. Kowalewski, "Cycle-Bounded Model Checking of PLC Software via Dynamic Large-Block Encoding", in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, H. M. Haddad, R. L. Wainwright, and R. Chbeir, Eds., ACM, 2018, pp. 1891–1898. DOI: 10.1145/3167132.3167334. [Online]. Available: https://doi.org/10.1145/3167132.3167334.

[Boh & Kow 20] D. Bohlender and S. Kowalewski, "Leveraging Horn clause solving for compositional verification of PLC software", *Discret. Event Dyn. Syst.*, vol. 30, no. 1, pp. 1–24, 2020. DOI: 10.1007/s10626-019-00296-8. [Online]. Available: https://doi.org/10.1007/s10626-019-00296-8.

[Böh & Pha+ 17] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, 2017, pp. 2329–2344. DOI: 10.1145/3133956.3134020. [Online]. Available: https://doi.org/10.1145/3133956.3134020.

[Böh & Roy+ 13] M. Böhme, A. Roychoudhury, and B. C. d. S. Oliveira, "Regression Testing of Evolving Programs", *Adv. Comput.*, vol. 89, pp. 53–88, 2013. DOI: 10.1016/B978-0-12-408094-2.00002-

3. [Online]. Available: https://doi.org/10.1016/B978-0-12-408094-2.00002-3.

[Boh & Sim+ 16] D. Bohlender, H. Simon, N. Friedrich, S. Kowalewski, and S. Hauck-Stattelmann, "Concolic Test Generation for PLC programs using Coverage Metrics", in *13th International Workshop on Discrete Event Systems, WODES 2016, Xi'an, China, May 30 - June 1, 2016*, C. G. Cassandras, A. Giua, and Z. Li, Eds., IEEE, 2016, pp. 432–437. DOI: 10.1109/WODES.2016.7497884. [Online]. Available: https://doi.org/10.1109/WODES.2016.7497884.

[Boh 21] D. Bohlender, "Symbolic Methods for Formal Verification of Industrial Control Software", Ph.D. dissertation, RWTH Aachen University, Germany, 2021. [Online]. Available: https://publications.rwth-aachen.de/record/835546.

[Bor & Tre+ 21] T. Borangiu, D. Trentesaux, P. Leitão, O. Cardin, and S. Lamouri, Eds., *Service Oriented, Holonic and Multi-Agent Manufacturing Systems for Industry of the Future - Proceedings of SOHOMA 2020, Paris, France, 1-2 October 2020*, vol. 952, Studies in Computational Intelligence, Springer, 2021, ISBN: 978-3-030-69372-5. DOI: 10.1007/978-3-030-69373-2. [Online]. Available: https://doi.org/10.1007/978-3-030-69373-2.

[Bra & Buc+ 13] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, "Simple and Efficient Construction of Static Single Assignment Form", in *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, R. Jhala and K. D. Bosschere, Eds., ser. Lecture Notes in Computer Science, vol. 7791, Springer, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6. [Online]. Available: https://doi.org/10.1007/978-3-642-37051-9%5C_6.

[Bra & Dal+ 22] P. Brauner, M. Dalibor, M. Jarke, I. Kunze, I. Koren, G. Lakemeyer, M. Liebenberg, J. Michael, J. Pennekamp, C. Quix, B. Rumpe, W. M. P. van der Aalst, K. Wehrle, A. Wortmann, and M. Ziefle, "A Computer Science Perspective on Digital Transformation in Production", *ACM Trans. Internet Things*, vol. 3, no. 2, 15:1–15:32, 2022. DOI: 10.1145/3502265. [Online]. Available: https://doi.org/10.1145/3502265.

[Bre & Buc+ 19] C. Brecher, M. Buchsbaum, and S. Storms, "Control from the Cloud: Edge Computing, Services and Digital Shadow for Automation Technologies", in *International Conference on Robotics and Automation, ICRA 2019, Montreal, QC, Canada, May 20-24,*

2019, IEEE, 2019, pp. 9327–9333. DOI: 10.1109/ICRA.2019.8793488. [Online]. Available: https://doi.org/10.1109/ICRA.2019.8793488.

[Bre & Klo+ 17] C. Brecher, F. Klocke, and R. H. Schmitt, Eds., *Internet of Production für agile Unternehmen : AWK Aachener Werkzeugmaschinen-Kolloquium 2017, 18. bis 19. Mai*, 29. Aachener Werkzeugmaschinen-Kolloquium, Aachen (Germany), 18 May 2017 - 19 May 2017, Aachen: Apprimus Verlag, May 18, 2017, 496 Seiten, ISBN: 3-86359-512-2. [Online]. Available: https://publications.rwth-aachen.de/record/692152.

[Bun 16] Bundesministerium für Bildung und Forschung. "Industrie 4.0". (Jan. 2016), [Online]. Available: https://www.bmbf.de/bmbf/de/forschung/digitale-wirtschaft-und-gesellschaft/industrie-4-0/industrie-4-0.html (visited on 06/16/2022).

[Bur & Sen 08] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation", in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, IEEE Computer Society, 2008, pp. 443–446. DOI: 10.1109/ASE.2008.69. [Online]. Available: https://doi.org/10.1109/ASE.2008.69.

[Cad & Dun+ 08] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs", in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds., USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[Cad & Gan+ 08] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death", *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 10:1–10:38, 2008. DOI: 10.1145/1455518.1455522. [Online]. Available: https://doi.org/10.1145/1455518.1455522.

[Cad & Sen 13] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later", *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. DOI: 10.1145/2408776.2408795. [Online]. Available: https://doi.org/10.1145/2408776.2408795.

[Can & God 19] G. Candea and P. Godefroid, "Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances", in *Computing and Software Science - State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and

G. J. Woeginger, Eds., vol. 10000, Springer, 2019, pp. 505–531. DOI: 10.1007/978-3-319-91908-9_24. [Online]. Available: https://doi.org/10.1007/978-3-319-91908-9%5C_24.

[Cha & Ulb+ 19] S. Cha, M. Ulbrich, A. Weigl, B. Beckert, K. Land, and B. Vogel-Heuser, "On the Preservation of the Trust by Regression Verification of PLC software for Cyber-Physical Systems of Systems", in *17th IEEE International Conference on Industrial Informatics, INDIN 2019, Helsinki, Finland, July 22-25, 2019*, IEEE, 2019, pp. 413–418. DOI: 10.1109/INDIN41052.2019.8972210. [Online]. Available: https://doi.org/10.1109/INDIN41052.2019.8972210.

[Cla & Hen+ 18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-10575-8.

[Cla & Kro+ 04] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs", in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, K. Jensen and A. Podelski, Eds., ser. Lecture Notes in Computer Science, vol. 2988, Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15. [Online]. Available: https://doi.org/10.1007/978-3-540-24730-2%5C_15.

[Dar & Bla+ 15] D. Darvas, E. Blanco Vinuela, and B. Fernández Adiego, " PLCverif: A Tool to Verify PLC Programs Based on Model Checking Techniques", WEPGF092. 4 p, 2015. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF092. [Online]. Available: https://cds.cern.ch/record/2213507.

[Dar & Kin 78] J. A. Darringer and J. C. King, "Applications of Symbolic Execution to Program Testing", *Computer*, vol. 11, no. 4, pp. 51–60, 1978. DOI: 10.1109/C-M.1978.218139. [Online]. Available: https://doi.org/10.1109/C-M.1978.218139.

[Dar & Maj+ 16] D. Darvas, I. Majzik, and E. B. Viñuela, "Formal Verification of Safety PLC Based Control Software", in *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, E. Ábrahám and M. Huisman, Eds., ser. Lecture Notes in Computer Science, vol. 9681, Springer, 2016, pp. 508–522. DOI: 10.1007/978-3-319-33693-0_32. [Online]. Available: https://doi.org/10.1007/978-3-319-33693-0%5C_32.

[dMou & Bjø 08] L. M. de Moura and N. S. Bjørner, "Z3: An Efficient SMT Solver", in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, vol. 4963, Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[Eng & Run+ 10] E. Engström, P. Runeson, and M. Skoglund, "A Systematic Review on Regression Test Selection Techniques", *Inf. Softw. Technol.*, vol. 52, no. 1, pp. 14–30, 2010. DOI: 10.1016/j.infsof.2009.07.001. [Online]. Available: https://doi.org/10.1016/j.infsof.2009.07.001.

[Fel & Gre+ 14] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating Regression Verification", in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds., ACM, 2014, pp. 349–360. DOI: 10.1145/2642937.2642987. [Online]. Available: https://doi.org/10.1145/2642937.2642987.

[Fre 60] E. Fredkin, "Trie memory", *Commun. ACM*, vol. 3, no. 9, pp. 490–499, 1960. DOI: 10.1145/367390.367400. [Online]. Available: https://doi.org/10.1145/367390.367400.

[Gal & Lyl 91] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance", *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 751–761, 1991. DOI: 10.1109/32.83912. [Online]. Available: https://doi.org/10.1109/32.83912.

[God & Kla+ 05] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing", in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds., ACM, 2005, pp. 213–223. DOI: 10.1145/1065010.1065036. [Online]. Available: https://doi.org/10.1145/1065010.1065036.

[God & Lah+ 11] P. Godefroid, S. K. Lahiri, and C. Rubio-González, "Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation", in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, E. Yahav, Ed., ser. Lecture Notes in Computer Science, vol. 6887, Springer, 2011, pp. 112–128. DOI: 10.1007/978-3-6

42-23702-7_12. [Online]. Available: https://doi.org/10.1007/978-3-642-23702-7%5C_12.

[God & Lev+ 12] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox Fuzzing for Security Testing", *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012. DOI: 10.1145/2093548.2093564. [Online]. Available: https://doi.org/10.1145/2093548.2093564.

[God & Str 09] B. Godlin and O. Strichman, "Regression Verification", in *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, ACM, 2009, pp. 466–471. DOI: 10.1145/1629911.1630034. [Online]. Available: https://doi.org/10.1145/1629911.1630034.

[God 07] P. Godefroid, "Compositional Dynamic Test Generation", in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, M. Hofmann and M. Felleisen, Eds., ACM, 2007, pp. 47–54. DOI: 10.1145/1190216.1190226. [Online]. Available: https://doi.org/10.1145/1190216.1190226.

[Gro & Kow+ 19] M. Grochowski, S. Kowalewski, M. Buchsbaum, and C. Brecher, "Applying Runtime Monitoring to the Industrial Internet of Things", in *24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2019, Zaragoza, Spain, September 10-13, 2019*, IEEE, 2019, pp. 348–355. DOI: 10.1109/ETFA.2019.8869447. [Online]. Available: https://doi.org/10.1109/ETFA.2019.8869447.

[Gro & Sim+ 20] M. Grochowski, H. Simon, D. Bohlender, S. Kowalewski, A. Löcklin, T. Müller, N. Jazdi, A. Zeller, and M. Weyrich, "Formale Methoden für rekonfigurierbare cyber-physische Systeme in der Produktion", *Autom.*, vol. 68, no. 1, pp. 3–14, 2020. DOI: 10.1515/auto-2019-0115. [Online]. Available: https://doi.org/10.1515/auto-2019-0115.

[Gro & Völ+ 22a] M. Grochowski, M. Völker, and S. Kowalewski, "Automatic Test Suite Generation for PLC Software in the Internet of Production", in *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2022, pp. 1–8. DOI: 10.1109/ETFA52439.2022.9921726.

[Gro & Völ+ 22b] M. Grochowski, M. Völker, and S. Kowalewski, "Test Suite Augmentation for Reconfigurable PLC Software in the Internet of Production", in *Formal Methods for Industrial Critical Systems - 27th International Conference, FMICS 2022, Warsaw, Poland, September 14-15, 2022, Proceedings*, J. F. Groote and M. Huisman, Eds., ser. Lecture Notes in Computer Science, vol. 13487,

Springer, 2022, pp. 137–154. DOI: 10.1007/978-3-031-15008-1_10. [Online]. Available: https://doi.org/10.1007/978-3-031-15008-1%5C_10.

[Guo & Kus+ 16] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental Symbolic Execution of Concurrent Software", in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 531–542. DOI: 10.1145/2970276.2970332. [Online]. Available: https://doi.org/10.1145/2970276.2970332.

[Gur & Cha 10] A. Gurfinkel and S. Chaki, "Boxes: A Symbolic Abstract Domain of Boxes", in *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, R. Cousot and M. Martel, Eds., ser. Lecture Notes in Computer Science, vol. 6337, Springer, 2010, pp. 287–303. DOI: 10.1007/978-3-642-15769-1_18. [Online]. Available: https://doi.org/10.1007/978-3-642-15769-1%5C_18.

[Gur & Kah+ 15] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn Verification Framework", in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, D. Kroening and C. S. Pasareanu, Eds., ser. Lecture Notes in Computer Science, vol. 9206, Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4%5C_20.

[Gyo & Lah+ 16] A. Gyori, S. K. Lahiri, and N. Partush, "Interprocedural Semantic Change-Impact Analysis using Equivalence Relations", *CoRR*, vol. abs/1609.08734, 2016. arXiv: 1609.08734. [Online]. Available: http://arxiv.org/abs/1609.08734.

[Gyo & Lah+ 17] A. Gyori, S. K. Lahiri, and N. Partush, "Refining Interprocedural Change-Impact Analysis using Equivalence Relations", in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, T. Bultan and K. Sen, Eds., ACM, 2017, pp. 318–328. DOI: 10.1145/3092703.3092719. [Online]. Available: https://doi.org/10.1145/3092703.3092719.

[IEC 20] IEC TR 62541-1:2020 RLV, "OPC unified architecture - Part 1: Overview and concepts", English, International Electrotechnical Commision, Standard, Nov. 18, 2020.

[Int 14] International Electrotechnical Commission, "IEC 61131: Programmable controllers - Part 3: Programming languages", International Electrotechnical Commission, Geneva, Switzerland,

Tech. Rep., 2014. DOI: 10.31030/2101412. [Online]. Available: https://dx.doi.org/10.31030/2101412.

[ISO 22] ISO/IEC/IEE 14764:2022, "Software engineering - Software life cycle processes - Maintenance", International Organization for Standardization, Tech. Rep., Jan. 2022. [Online]. Available: https://www.iso.org/standard/80710.html.

[Jam & Fra+ 13] K. Jamrozik, G. Fraser, N. Tillmann, and J. de Halleux, "Generating Test Suites with Augmented Dynamic Symbolic Execution", in *Tests and Proofs - 7th International Conference, TAP@STAF 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, M. Veanes and L. Viganò, Eds., ser. Lecture Notes in Computer Science, vol. 7942, Springer, 2013, pp. 152–167. DOI: 10.1007/978-3-642-38916-0_9. [Online]. Available: https://doi.org/10.1007/978-3-642-38916-0%5C_9.

[Jes & Bre+ 17] S. Jeschke, C. Brecher, T. Meisen, D. Özdemir, and T. Eschert, "Industrial Internet of Things and Cyber Manufacturing Systems", in *Industrial Internet of Things: Cybermanufacturing Systems*. Cham: Springer International Publishing, 2017, pp. 3–19, ISBN: 978-3-319-42559-7. DOI: 10.1007/978-3-319-42559-7_1. [Online]. Available: https://doi.org/10.1007/978-3-319-42559-7_1.

[Kin 76] J. C. King, "Symbolic Execution and Program Testing", *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. DOI: 10.1145/360248.360252. [Online]. Available: https://doi.org/10.1145/360248.360252.

[Kro & Tau 14] D. Kroening and M. Tautschnig, "CBMC - C Bounded Model Checker - (Competition Contribution)", in *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, E. Ábrahám and K. Havelund, Eds., ser. Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8%5C_26.

[Kuc & Pal+ 18] T. Kuchta, H. Palikareva, and C. Cadar, "Shadow Symbolic Execution for Testing Software Patches", *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, 10:1–10:32, 2018. DOI: 10.1145/3208952. [Online]. Available: https://doi.org/10.1145/3208952.

[Kuc 16] T. Kuchta, "Enhanced Symbolic Execution for Patch Testing and Document Recovery", Ph.D. dissertation, Imperial College London, UK, 2016. [Online]. Available: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.739615.

[Kuz & Kin+ 12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution", in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds., ACM, 2012, pp. 193–204. DOI: 10.1145/2254064.2254088. [Online]. Available: https://doi.org/10.1145/2254064.2254088.

[Kuz & Kin+ 14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution", in *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Germany*, W. Hasselbring and N. C. Ehmke, Eds., ser. LNI, vol. P-227, GI, 2014, pp. 45–46. [Online]. Available: https://dl.gi.de/20.500.12116/30951.

[Lah & Haw+ 12] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs", in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, P. Madhusudan and S. A. Seshia, Eds., ser. Lecture Notes in Computer Science, vol. 7358, Springer, 2012, pp. 712–717. DOI: 10.1007/978-3-642-31424-7_54. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7%5C_54.

[Leu & Whi 89] H. K. N. Leung and L. J. White, "Insights into Regression Testing", in *Proceedings of the Conference on Software Maintenance, ICSM 1989, Miami, FL, USA, 16-19 October, 1989*, IEEE, 1989, pp. 60–69. DOI: 10.1109/ICSM.1989.65194. [Online]. Available: https://doi.org/10.1109/ICSM.1989.65194.

[Lin & Mil+ 15] Y. Lin, T. Miller, and H. Søndergaard, "Compositional Symbolic Execution Using Fine-Grained Summaries", in *24th Australasian Software Engineering Conference, ASWEC 2015, Adelaide, SA, Australia, September 28 - October 1, 2015*, IEEE Computer Society, 2015, pp. 213–222. DOI: 10.1109/ASWEC.2015.32. [Online]. Available: https://doi.org/10.1109/ASWEC.2015.32.

[Lin & Mil+ 16] Y. Lin, T. Miller, and H. Søndergaard, "Compositional Symbolic Execution: Incremental Solving Revisited", in *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, A. Potanin, G. C. Murphy, S. Reeves, and J. Dietrich, Eds., IEEE Computer Society, 2016, pp. 273–

280. DOI: 10.1109/APSEC.2016.046. [Online]. Available: https://doi.org/10.1109/APSEC.2016.046.

[Lin 17] Y. Lin, "Symbolic Execution with Over-Approximation", Ph.D. dissertation, University of Melbourne, Parkville, Victoria, Australia, 2017. [Online]. Available: http://hdl.handle.net/11343/197985.

[Lop & Tou+ 22] I. D. Lopez-Miguel, J. Tournier, and B. F. Adiego, "PLCverif: status of a formal verification tool for programmable logic controller", *CoRR*, vol. abs/2203.17253, 2022. DOI: 10.48550/arXiv.2203.17253. arXiv: 2203.17253. [Online]. Available: https://doi.org/10.48550/arXiv.2203.17253.

[Ma & Kho+ 11] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed Symbolic Execution", in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, E. Yahav, Ed., ser. Lecture Notes in Computer Science, vol. 6887, Springer, 2011, pp. 95–111. DOI: 10.1007/978-3-642-23702-7_11. [Online]. Available: https://doi.org/10.1007/978-3-642-23702-7%5C_11.

[Mat 10] J. Matevska, "Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit", Ph.D. dissertation, Carl von Ossietzky University of Oldenburg, 2010, ISBN: 978-3-8348-1001-4. [Online]. Available: https://d-nb.info/999240757.

[Nie & Nie 20] F. Nielson and H. R. Nielson, "Program Analysis (an Appetizer)", *CoRR*, vol. abs/2012.10086, 2020. arXiv: 2012.10086. [Online]. Available: https://arxiv.org/abs/2012.10086.

[Nie & Nie 92] H. R. Nielson and F. Nielson, *Semantics with Applications - A Formal Introduction* (Wiley professional computing). Wiley, 1992, ISBN: 978-0-471-92980-2.

[Nie & Nie+ 99] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999, ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. [Online]. Available: https://doi.org/10.1007/978-3-662-03811-6.

[Nol & Ngu+ 19] Y. Noller, H. L. Nguyen, M. Tang, T. Kehrer, and L. Grunske, "Complete Shadow Symbolic Execution with Java PathFinder", *ACM SIGSOFT Softw. Eng. Notes*, vol. 44, no. 4, pp. 15–16, 2019. DOI: 10.1145/3364452.33644558. [Online]. Available: https://doi.org/10.1145/3364452.33644558.

[Nol 20] Y. Noller, "Hybrid Differential Software Testing", Ph.D. dissertation, Humboldt University of Berlin, Germany, 2020. [Online]. Available: http://edoc.hu-berlin.de/18452/22727.

[Pal & Kuc+ 16] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a Doubt: Testing for Divergences Between Software Versions", in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds., ACM, 2016, pp. 1181–1192. DOI: 10.1145/2884781.2884845. [Online]. Available: https://doi.org/10.1145/2884781.2884845.

[Pel 03] C. Peltz, "Web Services Orchestration and Choreography", *Computer*, vol. 36, no. 10, pp. 46–52, 2003. DOI: 10.1109/MC.2003.1236471. [Online]. Available: https://doi.org/10.1109/MC.2003.1236471.

[Pen & Gle+ 19] J. Pennekamp, R. Glebke, M. Henze, T. Meisen, C. Quix, R. Hai, L. C. Gleim, P. Niemietz, M. Rudack, S. Knape, A. Epple, D. Trauth, U. Vroomen, T. Bergs, C. Brecher, A. Bührig-Polaczek, M. Jarke, and K. Wehrle, "Towards an Infrastructure Enabling the Internet of Production", in *IEEE International Conference on Industrial Cyber Physical Systems, ICPS 2019, Taipei, Taiwan, May 6-9, 2019*, IEEE, 2019, pp. 31–37. DOI: 10.1109/ICPHYS.2019.8780276. [Online]. Available: https://doi.org/10.1109/ICPHYS.2019.8780276.

[Per & Dwy+ 08] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential Symbolic Execution", in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, M. J. Harrold and G. C. Murphy, Eds., ACM, 2008, pp. 226–237. DOI: 10.1145/1453101.1453131. [Online]. Available: https://doi.org/10.1145/1453101.1453131.

[Per & Yan+ 11] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution", in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds., ACM, 2011, pp. 504–515. DOI: 10.1145/1993498.1993558. [Online]. Available: https://doi.org/10.1145/1993498.1993558.

[PLC 08] PLCopen - Technical Committee 5, "Safety Software, Technical Specification, Part 2: User Examples", PLCopen, Tech. Rep., 2008.

[PLC 18] PLCopen - Technical Committee 5, "Safety Software, Technical Specification, Part 1: Concepts and Function Blocks for Safety Functions", PLCopen, Tech. Rep., 2018.

[Pod & Cla 90] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance", *IEEE Trans. Software Eng.*, vol. 16, no. 9, pp. 965–979, 1990. DOI: 10.1109/32.58784. [Online]. Available: https://doi.org/10.1109/32.58784.

[Rei & Kre+ 08] G. Reinhart, P. Krebs, and H. Schellmann, "Flexibilität und Wandlungsfähigkeit - das richtige Maß finden", *Institut für Werkzeugmaschinen und Betriebswissenschaften, TU München (Hg.): Innovationen für die Produktion. Landsberg/Lech: Moderne Industrie*, 2008.

[Rep & Hor+ 95] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability", in *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, R. K. Cytron and P. Lee, Eds., ACM Press, 1995, pp. 49–61. DOI: 10.1145/199448.199462. [Online]. Available: https://doi.org/10.1145/199448.199462.

[Rie 12] M. V. Riegen, "Ablaufkontrolle von Prozess-Choreographien", Ph.D. dissertation, University of Hamburg, 2012. [Online]. Available: http://ediss.sub.uni-hamburg.de/volltexte/2012/5852/.

[Rot & Har 96] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Trans. Software Eng.*, vol. 22, no. 8, pp. 529–551, 1996. DOI: 10.1109/32.536955. [Online]. Available: https://doi.org/10.1109/32.536955.

[Rot 96] G. Rothermel, "Efficient, Effective Regression Testing Using Safe Test Selection Techniques", AAI9703440, Ph.D. dissertation, USA, 1996, ISBN: 0591098520.

[Run & Per+ 12] N. Rungta, S. Person, and J. Branchaud, "A Change Impact Analysis to Characterize Evolving Program Behaviors", in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, IEEE Computer Society, 2012, pp. 109–118. DOI: 10.1109/ICSM.2012.6405261. [Online]. Available: https://doi.org/10.1109/ICSM.2012.6405261.

[San & Chi+ 08] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-Suite Augmentation for Evolving Software", in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, IEEE Computer Society, 2008, pp. 218–227. DOI: 10.1109/ASE.2008.32. [Online]. Available: https://doi.org/10.1109/ASE.2008.32.

[San & Har 10] R. A. Santelices and M. J. Harrold, "Exploiting Program Dependencies for Scalable Multiple-Path Symbolic Execution", in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, P. Tonella and A. Orso, Eds., ACM, 2010, pp. 195–206. DOI: 10.1145/1831708.1831733. [Online]. Available: https://doi.org/10.1145/1831708.1831733.

[San & Xi+ 21] M. Sanders, T. Xi, P. Dahlem, M. Fey, R. H. Schmitt, and C. Brecher, "On-Machine Measurements im Internet of Production", *Zeitschrift für wirtschaftlichen Fabrikbetrieb*, vol. 116, no. 4, pp. 259–262, 2021. DOI: doi:10.1515/zwf-2021-0037. [Online]. Available: https://doi.org/10.1515/zwf-2021-0037.

[Sen & Mar+ 05] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C", in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds., ACM, 2005, pp. 263–272. DOI: 10.1145/1081706.1081750. [Online]. Available: https://doi.org/10.1145/1081706.1081750.

[Sen & Nec+ 15] K. Sen, G. C. Necula, L. Gong, and W. Choi, "MultiSE: Multi-path Symbolic Execution using Value Summaries", in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 842–853. DOI: 10.1145/2786805.2786830. [Online]. Available: https://doi.org/10.1145/2786805.2786830.

[Sim & Fri+ 15] H. Simon, N. Friedrich, S. Biallas, S. Hauck-Stattelmann, B. Schlich, and S. Kowalewski, "Automatic Test Case Generation for PLC Programs using Coverage Metrics", in *20th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2015, Luxembourg, September 8-11, 2015*, IEEE, 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301602. [Online]. Available: https://doi.org/10.1109/ETFA.2015.7301602.

[Sim & Kow 18] H. Simon and S. Kowalewski, "Mode-Aware Concolic Testing for PLC Software - Special Session 'Formal Methods for the Design and Analysis of Automated Production Systems'", in *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*, C. A. Furia and K. Winter, Eds., ser. Lecture Notes in Computer Science, vol. 11023, Springer, 2018, pp. 367–376. DOI: 10.1007

/978-3-319-98938-9_21. [Online]. Available: https://doi.org/10.1007/978-3-319-98938-9%5C_21.

[Ste & Hei 17] T. Stehle and U. Heisel, "Konfiguration und Rekonfiguration von Produktionssystemen", in *Neue Entwicklungen in der Unternehmensorganisation*, D. Spath, E. Westkämper, H.-J. Bullinger, and H.-J. Warnecke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 333–367, ISBN: 978-3-662-55426-5. DOI: 10.1007/978-3-662-55426-5_39. [Online]. Available: https://doi.org/10.1007/978-3-662-55426-5_39.

[Ste 93] B. Steffen, "Generating Data Flow Analysis Algorithms from Modal Specifications", *Sci. Comput. Program.*, vol. 21, no. 2, pp. 115–139, 1993. DOI: 10.1016/0167-6423(93)90003-8. [Online]. Available: https://doi.org/10.1016/0167-6423(93)90003-8.

[Thö & Rei+ 17] D. Thönnessen, N. Reinker, S. Rakel, and S. Kowalewski, "A Concept for PLC Hardware-in-the-loop Testing Using an Extension of Structured Text", in *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*, IEEE, 2017, pp. 1–8. DOI: 10.1109/ETFA.2017.8247580. [Online]. Available: https://doi.org/10.1109/ETFA.2017.8247580.

[Thö & Sma+ 19] D. Thönnessen, N. Smallbone, M. Fabian, K. Claessen, and S. Kowalewski, "Testing Safety PLCs Using QuickCheck", in *15th IEEE International Conference on Automation Science and Engineering, CASE 2019, Vancouver, BC, Canada, August 22-26, 2019*, IEEE, 2019, pp. 1–6. DOI: 10.1109/COASE.2019.8843227. [Online]. Available: https://doi.org/10.1109/COASE.2019.8843227.

[Thö 21] D. Thönnessen, "Hardware-in-the-Loop Testing of Industrial Automation Systems Using PLC Languages", Ph.D. dissertation, RWTH Aachen University, Germany, 2021. [Online]. Available: https://publications.rwth-aachen.de/record/826036.

[Tie & Joh 09] M. Tiegelkamp and K. H. John, *SPS-Programmierung mit IEC 61131-3*. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00269-4. [Online]. Available: https://doi.org/10.1007/978-3-642-00269-4.

[Ule & Vog 16] S. Ulewicz and B. Vogel-Heuser, "System Regression Test Prioritization in Factory Automation: Relating Functional System Tests to the Tested Code using Field Data", in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society,*

Florence, Italy, October 23-26, 2016, IEEE, 2016, pp. 4619–4626. DOI: 10.1109/IECON.2016.7792997. [Online]. Available: https://doi.org/10.1109/IECON.2016.7792997.

[Ule & Vog 18] S. Ulewicz and B. Vogel-Heuser, "Industrially Applicable System Regression Test Prioritization in Production Automation", *IEEE Trans Autom. Sci. Eng.*, vol. 15, no. 4, pp. 1839–1851, 2018. DOI: 10.1109/TASE.2018.2810280. [Online]. Available: https://doi.org/10.1109/TASE.2018.2810280.

[Ule & Vog+ 17] S. Ulewicz, B. Vogel-Heuser, H. Simon, D. Bohlender, M. Obster, and S. Kowalewski, "A Priori Test Coverage Estimation for Automated Production Systems: Using Generated Behavior Models for Coverage Calculation", in *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*, IEEE, 2017, pp. 1–4. DOI: 10.1109/ETFA.2017.8247704. [Online]. Available: https://doi.org/10.1109/ETFA.2017.8247704.

[Ule 18] S. Ulewicz, "Test Coverage Assessment for Semi-Automatic System Testing and Regression Testing Support in Production Automation", Ph.D. dissertation, Technische Universität München, 2018.

[VDI 21] VDI/VDE 2206, "Entwicklung mechatronischer und cyberphysischer Systeme", VDI/VDE, Tech. Rep., Nov. 2021.

[Vog & Bou+ 18] B. Vogel-Heuser, S. Bougouffa, and M. Sollfrank, "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the extended Pick and Place Unit", Institute of Automation and Information Systems, Technische Universität München, Tech. Rep., 2018.

[Vog & Fol+ 14] B. Vogel-Heuser, J. Folmer, and C. Legat, "Anforderungen an die Softwareevolution in der Automatisierung des Maschinen- und Anlagenbaus", *Autom.*, vol. 62, no. 3, pp. 163–174, 2014. DOI: 10.1515/auto-2013-1051. [Online]. Available: https://doi.org/10.1515/auto-2013-1051.

[Vog & Leg+ 14] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann, "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit", Institute of Automation and Information Systems, Technische Universität München, Tech. Rep., 2014.

[Vog & Rös+ 16] B. Vogel-Heuser, S. Rösch, J. Fischer, T. Simon, S. Ulewicz, and J. Folmer, "Fault Handling in PLC-Based Industry 4.0 Automated Production Systems as a Basis for Restart and Self-Configuration and Its Evaluation", *Journal of Software Engi-*

neering and Applications, vol. 09, no. 01, pp. 1–43, 2016. DOI: 10.4236/jsea.2016.91001. [Online]. Available: https://doi.org/10.4236/jsea.2016.91001.

[Wei & Ulb+ 20] A. Weigl, M. Ulbrich, and D. Lentzsch, "Modular Regression Verification for Reactive Systems", in *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 12477, Springer, 2020, pp. 25–43. DOI: 10.1007/978-3-030-61470-6_3. [Online]. Available: https://doi.org/10.1007/978-3-030-61470-6%5C_3.

[Wei & Wie+ 17] A. Weigl, F. Wiebe, M. Ulbrich, S. Ulewicz, S. Cha, M. Kirsten, B. Beckert, and B. Vogel-Heuser, "Generalized Test Tables: A Powerful and Intuitive Specification Language for Reactive Systems", in *15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017*, IEEE, 2017, pp. 875–882. DOI: 10.1109/INDIN.2017.8104887. [Online]. Available: https://doi.org/10.1109/INDIN.2017.8104887.

[Wei 21] A. Weigl, "Formal Specification and Verification for Automated Production Systems", Ph.D. dissertation, Karlsruhe Institute of Technology, Germany, 2021. [Online]. Available: https://nbn-resolving.org/urn:nbn:de:101:1-20211222040231640 80066.

[Wey & Jen 91] E. J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies", *IEEE Trans. Software Eng.*, vol. 17, no. 7, pp. 703–711, 1991. DOI: 10.1109/32.83906. [Online]. Available: https://doi.org/10.1109/32.83906.

[Wie & Rei+ 14] H.-P. Wiendahl, J. Reichardt, and P. Nyhuis, *Handbuch Fabrikplanung: Konzept, Gestaltung und Umsetzung wandlungsfähiger Produktionsstätten*. Carl Hanser Verlag GmbH Co KG, 2014.

[Xu & Kim+ 15] Z. Xu, Y. Kim, M. Kim, M. B. Cohen, and G. Rothermel, "Directed test suite augmentation: an empirical investigation", *Softw. Test. Verification Reliab.*, vol. 25, no. 2, pp. 77–114, 2015. DOI: 10.1002/stvr.1562. [Online]. Available: https://doi.org/10.1002/stvr.1562.

[Yan & Khu+ 13] G. Yang, S. Khurshid, and C. S. Pasareanu, "Memoise: A Tool for Memoized Symbolic Execution", in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl,

Eds., IEEE Computer Society, 2013, pp. 1343–1346. DOI: 10.1109/ICSE.2013.6606713. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606713.

[Yan & Per+ 14] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed Incremental Symbolic Execution", *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, 3:1–3:42, 2014. DOI: 10.1145/2629536. [Online]. Available: https://doi.org/10.1145/2629536.

[Yoo & Har 12] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation: A Survey", *Softw. Test. Verification Reliab.*, vol. 22, no. 2, pp. 67–120, 2012. DOI: 10.1002/stv.430. [Online]. Available: https://doi.org/10.1002/stv.430.

[Zel & Jaz+ 18] A. Zeller, N. Jazdi, and M. Weyrich, "Verifikation verteilter Automatisierungssysteme auf Basis einer Modellkomposition", *Autom.*, vol. 66, no. 6, pp. 456–470, 2018. DOI: 10.1515/auto-2017-0069. [Online]. Available: https://doi.org/10.1515/auto-2017-0069.

[Zel & Jaz+ 19] A. Zeller, N. Jazdi, and M. Weyrich, "Functional verification of distributed automation systems", *The International Journal of Advanced Manufacturing Technology*, vol. 105, no. 9, pp. 3991–4004, 2019. DOI: 10.1007/s00170-019-03791-2. [Online]. Available: https://doi.org/10.1007/s00170-019-03791-2.

[Zel & Wey 15] A. Zeller and M. Weyrich, "Test Case Selection for Networked Production Systems", in *20th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2015, Luxembourg, September 8-11, 2015*, IEEE, 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301604. [Online]. Available: https://doi.org/10.1109/ETFA.2015.7301604.

[Zel & Wey 16] A. Zeller and M. Weyrich, "Challenges for Functional Testing of reconfigurable Production Systems", in *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*, IEEE, 2016, pp. 1–4. DOI: 10.1109/ETFA.2016.7733620. [Online]. Available: https://doi.org/10.1109/ETFA.2016.7733620.

[Zel & Wey 18] A. Zeller and M. Weyrich, "Composition of Modular Models for Verification of Distributed Automation Systems", *Procedia Manufacturing*, vol. 17, Jun. 2018. DOI: 10.1016/j.promfg.2018.10.139.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of (more than 580) reports dating back to 1987 is available from

http://aib.informatik.rwth-aachen.de/

or can be downloaded directly via

http://aib.informatik.rwth-aachen.de/tex-files/berichte.pdf

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2021-01	Mathias Obster: Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe
2021-02	Manfred Nagl: An Integrative Approach for Software Architectures
2021-03	Manfred Nagl: Sequences of Software Architectures
2021-04	Manfred Nagl: Embedded Systems: Simple Rules to Improve Adaptability
2021-05	Manfred Nagl: Process Interaction Diagrams are more than Process Chains or Transport Networks
2021-06	Manfred Nagl: Characterization of Shallow and Deep Reuse
2021-07	Martin Schweigler: Ground Surface Pattern Recognition for Enhanced Navigation
2021-08	Manfred Nagl: The Architecture is the Center of the Software Development Process
2021-09	Manfred Nagl: Architectural Styles: Do they Need Different Notations?
2021-10	David Thönnessen: Hardware-in-the-Loop testing of industrial automation systems using PLC languages
2021-11	Dimitri Bohlender: Symbolic Methods for Formal Verification of Industrial Control Software
2022-01	Helen Bolke-Hermanns, Klaus Indermark, Joost-Pieter Katoen, Stefan Kowalewski, Thomas Noll, and Wolfgang Thomas: 50 Jahre Studiengang Informatik an der RWTH — Ein Streifzug in Text und Bild
2022-02	Shahid Khan: Boolean-logic driven Markov processes : Explained. Analysed. Verified.
2023-01	Marcus Völker: Policy Iteration for Value Set Analysis of PLC Programs

2023-02 Maximilian Kloock: Synchronization-based cooperative trajectory planning of networked vehicles
2023-03 Thomas Noll and Ira Fesefeldt: 22. Kolloquium Programmiersprachen und Grundlagen der Programmierung
2023-04 Jera Hensel: Automated Termination Analysis of C Programs