Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Practice-Oriented Formal Methods to Support the Software Development of Industrial Control Systems

Ph.D. Dissertation

Dániel Darvas

Thesis supervisor:
István Majzik, Ph.D. (BUTE)

Advisor:
Enrique Blanco Viñuela, Ph.D. (CERN)

Budapest
2017

Dániel Darvas
http://mit.bme.hu/~darvas/

January 2017

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

H-1117 Budapest, Magyar tudósok körútja 2.

doi: 10.5281/zenodo.162950

Declaration of own work and references

I, Dániel Darvas, hereby declare that this dissertation, and all results claimed therein are my own work, and rely solely on the references given. All segments taken word-by-word, or in the same meaning from others have been clearly marked as citations and included in the references.

Nyilatkozat önálló munkáról, hivatkozások átvételéről

Alulírott Darvas Dániel kijelentem, hogy ezt a doktori értekezést magam készítettem és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2017. 01. 11.

Darvas Dániel

"I would therefore like to posit that computing's central challenge, viz. 'How not to make a mess of it', has *not* been met. On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence."

Edsger W. Dijkstra [Dij01]

"Funding agencies often require that larger research-funded projects [...] demonstrate the practicality of an approach on 'real' examples. When authors report such efforts, they state that they are successful. Paradoxically, such success stories reveal the failure of industry to adopt formal methods as standard procedures; if using these methods was routine, papers describing successful use would not be published."

David L. Parnas [Par10]

Acknowledgements

This dissertation concludes a work that has been started in 2009. I would like to express my gratitude towards everyone who helped me on this long journey:

- My family who supported me through this whole trip;
- My Ph.D. supervisors, István Majzik and Enrique Blanco Viñuela who trusted and supported me, and have guided my work during the last three years;
- My B.Sc. and M.Sc. supervisors, Tamás Bartha and András Vörös who oriented me towards research and helped me much more than any student could ever expect;
- My former and present colleagues and friends at the Fault Tolerant Systems Research Group of the Budapest University of Technology and Economics who provided continuous help, support and fun at work and outside of it;
- My former and present colleagues and friends at the European Organization for Nuclear Research (CERN);
- The PetriDotNet Team for all the improvements and memorable achievements that we have reached together;
- Kinga Györffy, Zoltán Gönye and Tamás Polyák who helped me to understand how to convey a message;
- All the friends who helped to keep me sane while doing research;
- The colleagues at the CERN visits service who let me exercise another passion and to clean my mind;
- The staff of the swimming pools in Ferney-Voltaire and Gex who kept running the facilities where many of the presented ideas were born.

Ágnes, Ákos, András, Attila, Bálint, Borja, Christina, Dávid, Gábor, James, Jesús, Josef, Kristóf, Łukasz, Matěj, Stefan, Tamás, Valentin, Vince, William, Zita, Zoltán, and everyone else I forgot to mention explicitly: thanks for all.

Special thanks to everyone who have read the drafts of my dissertation and provided me feedback, new ideas or corrections.

I would like to thank also the support of CERN, providing the funding of this Ph.D. research project through the Doctoral Student programme.

Summary

Formal specification and verification methods provide ways to describe requirements precisely and to check whether the requirements are satisfied by the design or the implementation. In other words, they can prevent development faults and therefore improve the quality of the developed systems. These methods are part of the state-of-the-practice in application domains with high criticality, such as avionics, railway or nuclear industry.

The situation is different in the industrial control systems domain. As the criticality of the systems is much lower, formal methods are rarely used. The two main obstacles to using formal methods in systems with low- or medium-criticality are *performance* and *usability*. Overcoming these obstacles often needs deep knowledge and high effort. Model checking, one of the main formal verification techniques, is computationally difficult, therefore the analysis of non-trivial systems requires special considerations. Furthermore, the mainly academic tools implementing different model checking algorithms are not suitable for users who are not experts in formal methods. The situation is similar with formal specification methods: they are typically too abstract or theoretical to be used by non-specialists, with reasonably long training period.

This work provides various solutions to both the challenges of performance and usability, and centred around the formal verification of industrial control systems. The aim is to provide more efficient verification algorithms and easy-to-use, practice-oriented formal (verification and specification) methods that can be applied to PLC (programmable logic controller) software used in industrial control systems, where the use of heavyweight, low-level methods is not necessary or not feasible. The proposed methods take the particularities of the target domain into account, making formal methods accessible without excessive effort needed.

First, this dissertation provides B-I-Sat, a new algorithm that improves the performance of the saturation-based model checking techniques by combining it with bounded model checking techniques. Saturation-based model checking already provides good performance for many different models. By combining it with bounded techniques, this performance can be further improved in certain cases.

Second, a verification workflow and its implementation are presented that allow the industrial practitioners to use the model checking for PLC-based control software. This is achieved by hiding all formal details and adapting the inputs and outputs of the verification workflow to the specific needs of the domain and the available knowledge. The contributions include a model checker-independent representation of the programs to be verified and property-preserving reduction algorithms to make the formal analysis feasible. Special attention is paid to the verification of safety-critical PLC programs, where development restrictions impose additional needs for the verification workflow.

Third, a formal specification language is proposed that is specifically targeting the behaviour description of program modules used in the PLC-based industrial control software. The language itself is heavily adapted to the domain and its needs. Furthermore, it is complemented by static analysis, code generation and conformance checking methods. For the conformance checking, special relations were introduced, responding to the real needs observed in the domain.

All these contributions are demonstrated and evaluated on real, industrial examples.

Összefoglaló

A formális specifikációs és verifikációs módszerek használatával lehetőség nyílik követelmények precíz leírására és annak ellenőrzésére, hogy a követelményeket kielégíti-e egy terv vagy megvalósítás. Más nézőpontból e módszerek segítségével elkerülhetők, illetve felfedhetők és javíthatók bizonyos fejlesztési hibák, ami által a fejlesztett rendszerek minősége javul. A formális módszerek használata mára elterjedt gyakorlattá vált a biztonságkritikus rendszerek, mint például légi, vasúti vagy nukleáris rendszerek fejlesztésében.

Az ipari vezérlőrendszerek területén mást tapasztalhatunk. Mivel e rendszerek jóval kevésbé kritikusak, a fejlesztés során ritka a formális módszerek használata. Ezen módszerek alacsony vagy közepes kritikusságú rendszerek fejlesztésében való alkalmazásának két fő akadálya a korlátozott *teljesítmény* és *használhatóság*, amelyek leküzdéséhez gyakran nagy szaktudás és erőfeszítés szükséges. A modellellenőrzés, az egyik gyakran használt formális verifikációs módszer meglehetősen számításigényes, így a nemtriviális rendszerek modellellenőrzése különleges technikákat igényel. Továbbá a modellellenőrző algoritmusokat megvalósító, főként akadémiai eszközök nehezen használhatók a formális verifikációban nem jártas felhasználók számára. Hasonló tapasztalható a formális specifikáció terén is: ezek a módszerek általában túl absztraktak vagy matematikaközeliek ahhoz, hogy nem specialisták komoly képzés nélkül használni tudják.

Jelen munka többféle megoldást nyújt mind a teljesítmény, mind a használhatóság kihívására, fókuszba helyezve az ipari vezérlőrendszereket. A munka célja hatékonyabb verifikációs algoritmusokat és könnyen használható, gyakorlatorientált formális (verifikációs és specifikációs) módszereket javasolni, amelyek felhasználhatók a programozható logikai vezérlők (PLC-k) szoftverének fejlesztésében, ahol a „nehézsúlyú”, alacsony szintű módszerek alkalmazása tipikusan nem lehetséges vagy nem szükséges. A javasolt módszerek figyelembe veszik a megcélzott szakterület sajátosságait, így elérhetővé teszik a formális módszerek használatát túlzott erőfeszítés nélkül.

Elsőként egy új verifikációs algoritmus, a B-I-Sat kerül bemutatásra, amely a szaturációalapú modellellenőrzési technikák teljesítményét javítja azáltal, hogy az algoritmust ötvözi a korlátos modellellenőrzési módszerekkel. A szaturációalapú modellellenőrzés már számos modellen bizonyította hatékony működését. Ennek a korlátos módszerekkel történő integrációja tovább javíthatja a teljesítményét bizonyos esetekben.

Másodikként egy verifikációs folyamat és annak megvalósítása kerül bemutatásra, amely lehetővé teszi az ipari fejlesztőknek a PLC-alapú vezérlőprogramok modellellenőrzését. Ez annak köszönhető, hogy a folyamat elrejti az összes, formális verifikáció terén speciális ismereteket igénylő részletet, és olyan be- és kimeneteket definiál, amelyek illeszkednek a szakterülethez és az elvárható ismeretszinthez. A bemutatott kontribúció magában foglalja a verifikálandó programok egy modellellenőrzőfüggetlen leírását és tulajdonságmegőrző redukciós algoritmusokat a hatékonyság növelése érdekében. Különleges figyelmet kap a biztonságkritikus PLC-programok ellenőrzése, ahol a fejlesztési folyamat megkötései befolyásolják a verifikációs folyamatot.

Harmadikként egy formális specifikációs nyelvet javaslok, amely kimondottan a PLC-alapú ipari vezérlőprogramokban használt programmodulok viselkedésének meghatározását célozza. A nyelv figyelembe veszi a szakterületet és annak igényeit. A specifikációs módszer kiegészítésre került statikus analízissel, valamint kódgeneráló és konformanciaellenőrző módszerekkel. A konformanciaellenőrzéshez új relációkat vezettem be, amelyek a szakterület valós, megfigyelt szükségleteihez illeszkednek.

Mindegyik új eredmény valódi, ipari példákon keresztül kerül bemutatásra és értékelésre.

Contents

Chapter 1

Introduction

1.1 Preliminaries and Objectives

Dependability[1] is an integrating concept comprising availability, reliability, safety, integrity and maintainability. This is a desired property, especially for critical systems. A *failure* is an observable deviation from the system's required behaviour, thus a threat to dependability. The cause of a failure is the propagation of an *error*, which itself is a certain (internal) system state that can result in a failure. The causes of the errors are the *faults* [Avi+04].

There are various means to attain the attributes of dependability: *fault prevention*, *fault tolerance*, *fault removal* and *fault forecasting* [Avi+04]. Formal methods are well-known techniques for the prevention of *development faults* and some of the *operational faults*, by providing mathematically sound, unambiguous means for the description and verification of the system's requirements [Mar94]. Formal verification and formal specification are getting more and more used in *safety-critical* application domains where the consequences of a failure are *catastrophic* [Avi+04; Woo+09]. This can be either because a single failure may cause an accident or loss of life, or it implies a high economic loss (e.g. in avionics [Sou+09], railway systems [LSP07], space applications [Hav+00]), or the undesired consequence affects a large number of systems (e.g. mass-produced processors [Fix08; Kai+09]) causing a high total cost.

Industrial control systems are used in various settings. If the functionality of a control system is safety-critical, the IEC 61508-2 standard [I61508-2] defines required development and verification methods for each *safety integrity level* (SIL), depending on the probability of tolerable hazards (hazardous failures). The industrial control systems consist of many components. Often a key element is a *programmable logic controller* (PLC): a robust, reconfigurable, specialised computer that performs the control tasks. In certain cases, their operation is safety-critical, but many times – due to additional safety-related systems or measures (e.g. physical barriers, independent safety relays) – the target SIL for the PLC-based controller is below SIL 1, the lowest SIL defined in IEC 61508. Even though in these cases the expected failure is rare or not catastrophic, this does not mean that a failure (e.g. an *outage*) cannot cause significant economic losses. However, the lower SIL typically manifests in reduced verification budget. In these cases the use of *heavyweight* formal methods (e.g. B Method or Z for specification; or manual use of theorem provers for verification) would need excessive effort. Besides the difficult usage (high training costs, need for special expertise), another common obstacle to using formal verification methods is their performance. For example, exhaustively checking the behaviour

[1] In the dissertation the taxonomy proposed by Avižienis, Laprie *et al.* [Avi+04] is used, which is briefly introduced and summarised in the following two paragraphs.

(state space) of a model is a computationally difficult task, therefore most of the algorithms cannot scale up to the size of industrial problems.

Goal. The main goal of this research is to analyse the applicability of formal methods in the domain of industrial control systems and to propose specification and verification methods. As mentioned above, the two main challenges of using these methods are *performance* and *usability*. This dissertation proposes various solutions to both challenges. It aims to provide more efficient verification algorithms and easy-to-use, practice-oriented formal (specification and verification) methods that can be applied to PLC software used in industrial control systems, where the use of heavyweight methods is not necessary or not feasible. The methods to be proposed should take the particularities of the target domain into account.

1.1.1 Introduction to Formal Verification

Verification is "[t]he process of evaluating a system or component to determine whether the products [...] satisfy the conditions imposed at the start" [I1012]. *Formal verification* techniques are mathematically sound methods to precisely determine the satisfaction of the given formalised requirements.

"*Model checking* is an automated [formal verification] technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model" [BK08]. As a more precise definition, model checking is a method to find all states in a model (given as Kripke structure) which satisfy a given temporal logic formula [Cla08]. This method was first described by Edmund M. Clarke and E. Allen Emerson in [CE82], also by Jean-Pierre Queille and Joseph Sifakis independently [QS82].

Being an automated method, it is a good candidate for lightweight formal verification, it has a potential to be a "push-button technology" whose usage does not require high degree of user interaction or expertise [BK08]. Furthermore, it can provide diagnostic traces (counterexamples or witnesses) that is useful feedback about the problems found.

Modelling and requirement description formalisms. While model checking operates on a Kripke structure according to its definition, typically higher-level modelling languages are used for representing the models, such as Petri nets [Mur89], (timed or untimed) automata [AD94], or process algebra [Fok00]. Similarly, multiple different temporal logic can be used to describe the requirements to check. The most frequently used ones are linear temporal logic (LTL) and computation tree logic (CTL). The different formalisms have different expressivity, also the model checking algorithms can differ significantly depending on the supported formalisms.

Model checking methods. E. Clarke, one of the creators of model checking said that the model checking algorithm is an "intelligent exhaustive search of the state space to determine if the specification is true or not" [CES09]. With the increase of the number of reachable states in a system, the "intelligence" of the state space exploration algorithms is getting more and more important. The so-called *explicit* methods represent each state of the given model individually. This allows to use simple algorithms from graph theory, but fails to provide solution for large models where the individual handling of each state is not possible. Large state sets (state spaces) may be caused by various reasons, e.g. the large number of inputs and outputs or concurrent behaviours. This is the well-known *state space explosion* problem. Over the years various solutions emerged to handle this issue:

- *Symbolic algorithms*, which store the state space in a more compact, encoded format, e.g. using decision diagrams;

- *Abstractions*, which simplify the model to have a smaller state space;
- *Bounded algorithms*, which limit the depth of the state space exploration (from the initial state) to reduce the size of the explored state space.

The first symbolic algorithms were based on binary decision diagrams [Bur+92]. Since then new algorithms were developed, using different exploration strategies and data structures. One of the promising solutions is *saturation* [CLS01], which "tends to perform extremely well when applied to discrete-event systems having multiple asynchronous events that depend and affect only relatively small subsystems" [CZJ12]. The fact that the algorithm "performs well" means that the set of verifiable models and requirements is larger or the execution time is shorter, but it does not mean that there are no limitations imposed by the performance needs of the algorithm. In certain cases excessive amount of memory is required to perform the verification.

There is no silver bullet for model checking, each method has its disadvantages and limitations. Some approaches tried already to combine ideas from different methods, e.g. [Cha+02; Cop+01; CNQ05]. A possible improvement of the saturation-based model checking is to combine it with bounded model checking, which – to the author's best knowledge – was not studied before this research project. This could also help the verification of industrial control systems by improving the earlier verification performance (e.g. compared to [*c*28]). Evaluation of this possibility is a challenge of this dissertation (Challenge 1).

Model checking as a part of the PLC software development process. Model checking has already proven to be useful in various domains [Cla08]. However, providing the necessary formal models and the requirements as temporal logic formulae is a difficult task, especially for the people not familiar with formal methods. Furthermore, model checking may also need manual adaptation, fine-tuning to the current problem to improve the performance. This implies a high cost of usage, which may be an obstacle to apply model checking in the development of PLC programs.

The academic algorithm design and development efforts led to high-performance general-purpose model checkers (e.g. UPPAAL [Amn+01], LTSmin [Kan+15], NuSMV/nuXmv [Cav+14]). However, general-purpose tools cannot improve the domain-specific usability of the verification method. To improve the usability and to integrate formal verification into the industrial control system development processes, the focus should be specifically set to this domain.

Although the use of model checking for PLC-based industrial control software was already studied in e.g. [GSF08; SD08; BBK12], these works did not provide generic solutions applicable in real-life, or this aspect was not emphasised. Making model checking adapted to the PLC program development domain, usable directly by the PLC developers; and making it scalable are challenges of this dissertation (Challenges 2, 3). Furthermore, special attention should be paid to a special branch of PLCs, the so-called *fail-safe* or *safety PLCs*. To attain a high level of confidence during the software development process for such PLCs, special restrictions and development methods are followed (e.g. coding conventions, programming language restrictions), these have to be taken into account for the solution to be proposed (Challenge 4).

1.1.2 Introduction to Formal Specification

Requirements engineering is a set of activities to explore, evaluate and document the objectives, capabilities, constraints and assumptions of a system to be designed [Lam09]. The *(requirements) specification* is the act of "detailing, structuring and documenting the agreed characteristics of the system-to-be" [Lam09]. According to [Lam00], *formal specification* "is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy." One of the

facts making the formal specification process difficult is that "[s]pecifications are never formal in the first place" and they are "hard to develop and assess" [Lam00]. The expected benefits of formalising the specification method are "a higher degree of precision in the formulation [...], precise rules for their interpretation and much more sophisticated forms of validation and verification" [Lam09].

Formal specification is studied since the late 1960s, and since then several methods emerged. Petri nets [Mur89], Lotos [I8807], the B Method [Abr96], Z [I13568], or the communicating sequential processes (CSP) [Hoa85] are widely-known techniques. Though widely-known, they are not widely used in the industry [Kni+97], because they are too complex, they need too deep mathematical knowledge, or their abstraction level is too high. Therefore the usage of these specification methods is restricted to highly critical domains, e.g. avionics [HLR98] or railway industry [But02].

In the industrial control systems domain, the state-of-the-art development processes still rely on informal specifications and hidden assumptions. These specifications are often ambiguous, leading to misunderstanding and unintended behaviours in the implementation. The lack of unambiguous specification imposes a problem for the formal verification too: how can we decide if the implementation is correct, if we do not know what is correctness, i.e. what are the expected properties?

There are various attempts to provide better, PLC-specific specification methods, e.g. [Lju+10; Luk+13]. Analysing the existing methods and finding a suitable specification is a challenge of this work (Challenge 5). It is another challenge to provide formal verification for PLC software on the basis of the selected specification method (Challenge 6).

1.1.3 Summary of New Challenges

Challenge 1: Designing model checking algorithms combining bounded and saturation-based techniques to improve their performance. Both bounded model checking and saturation-based techniques increase the set of models on which verification is feasible compared to basic explicit model checking algorithms. Is it possible to combine these two approaches? Does it improve the performance with respect to the original saturation-based model checking?

Challenge 2: Making model checking easily accessible to the PLC developers. Model checking is rarely used for industrial control software, mainly because of the enormous effort needed to create formal models and requirements, furthermore to learn the usage of the model checker tools. How can model checking be made accessible and practically applicable in the PLC program development process? How can model checking be used without excessive effort, without exposing the users (PLC developers) to complex mathematical formalisms?

Challenge 3: Making the PLC model checking applicable to real-world PLC programs. The formal models of real PLC modules or programs and their state spaces tend to be extremely large, making the model checking infeasible using general-purpose model checker tools. Could heuristic model reductions reduce the performance needs of model checking and therefore cope with a bigger set of models?

Challenge 4: Extending the model checking approach to safety-critical PLC programs. The original PLC model checking workflow supported the Siemens SCL language only, which – being a high-level language – is more suitable for the implementation of complex programs. However, the development of PLCs used in safety-critical settings has specific procedures and restrictions, such as the mandatory usage of FBD or LAD languages (in case of Siemens PLCs). How can model checking be adapted to these lower-level programming

languages used in safety-critical PLC program development?

Challenge 5: Providing lightweight formal specification for PLC software modules. Unambiguous requirements are essential for any development or verification activity. Formal specifications may reduce the ambiguity, but the general-purpose formal specification methods are too complex and non-intuitive to be used in the PLC development domain with a reasonable effort. What are the requirements towards a formal specification language specially adapted to the PLC domain? What formal specification method can aid the PLC program development process?

Challenge 6: Providing verification solutions based on formal specification. How could formal specification improve the PLC program verification? What verification methods can be used to check the conformance between a PLC program and its formal specification? How can this be made useful in practice, without excessive amount of false positives (i.e. without having a high number of detected differences that are considered to be acceptable by the developers)?

These challenges led to a research project with new scientific results in three different areas: a new, saturation-based bounded model checking algorithm with different iteration strategies (Thesis 1), a new method to apply model checking for PLC programs (Thesis 2) and the definition of a formal specification language for PLC software modules together with its application methods (Thesis 3). Table 1.1 presents the correspondence between the discussed challenges and the results of this work.

Table 1.1: Correspondence between the discussed challenges and the proposed solutions

		Challenge					
		1	2	3	4	5	6
Thesis 1	*Chapter 2 of the dissertation*	•					
Thesis 2	*Chapter 3 of the dissertation*		•	•	•		
Thesis 3	*Chapter 4 of the dissertation*					•	•

1.2 Contributions and Structure of the Dissertation

This dissertation presents three different contributions. These contributions are centred around formal verification of industrial control software. The challenges are twofold: they target the performance and the usability of model checking that are two major obstacles of using model checking in real life scenarios. The high-level challenges and the contributions are summarised in Figure 1.1.

- *Thesis 1* (Chapter 2) discusses the improvement of saturation, a model checking algorithm which provided good performance for many different models. In this thesis the combination of saturation with bounded model checking will be presented. This new approach may further improve the performance of the saturation algorithms, increasing the set of requirements and models that are possible to be verified. This general method may give a solution for industrial control software when it is not efficient to develop a dedicated workflow for the given use case.

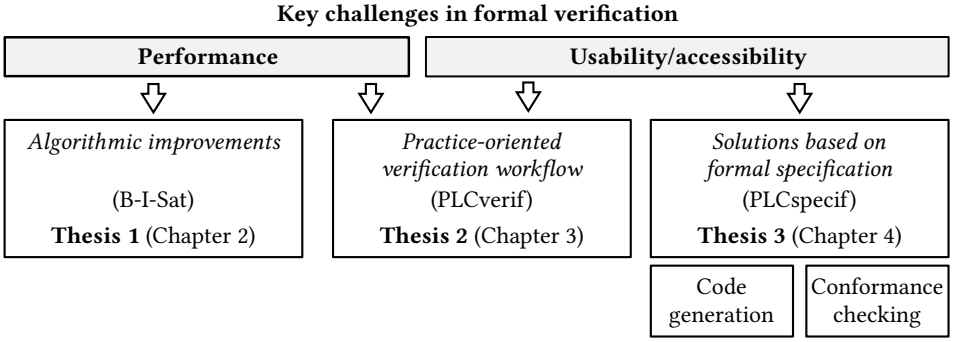


Figure 1.1: Overview of the contributions of the dissertation

Here the focus is on the performance improvements, thus it may be an appropriate solution when the performance is the bottleneck of the verification, or when it is not efficient to design and implement a dedicated verification workflow for the given verification case(s).

This thesis responds to Challenge 1 discussed in Section 1.1.3.

- *Thesis 2* (Chapter 3) focuses on the usability aspects of model checking applied to PLC-based industrial control software. It provides a verification workflow that makes model checking of PLC programs directly usable by the developers, without requiring extensive training, direct modelling or manual reductions. The presented contributions are related mainly to *(i)* the intermediate model language used for the model checker-independent representation of the PLC programs, *(ii)* the reduction heuristics that reduce efficiently the size of the intermediate model representations, *(iii)* the extension of the verification workflow to support the development and verification of safety-critical PLC programs, and finally *(iv)* the implementation and the evaluation of the verification workflow.

  This part focuses on making model checking of PLC programs accessible to developers. The verification workflow is flexible and it only needs the PLC source code and informal requirements, which are formalised using given requirement patterns by the user.

  This thesis responds to Challenges 2, 3 and 4 discussed in Section 1.1.3.

- *Thesis 3* (Chapter 4) is dedicated to PLCspecif, a novel complete, formal behaviour specification language specifically targeting PLC modules. Besides the syntax and semantics definition, it is extended with code generation, invariant checking, static analysis and conformance checking facilities, to provide wide support for the development and analysis of PLC software.

  This contribution, similarly to Thesis 2, also focuses on the usability improvements of the PLC program verification, but here a more thorough analysis is targeted. This additionally requires the formal specification of the module to be verified, but the benefit is a deeper analysis and the possibility of automated implementation generation.

  This thesis responds to Challenges 5 and 6 discussed in Section 1.1.3.

Chapter 2

Bounded Model Checking Based on Saturation

Model checking is a successful verification technique, however its usage is limited partially because of its performance. Various solutions emerged during the last 30 years to improve the performance. The so-called saturation algorithms successfully pushed the limits of model checking and increased the set of verifiable problems. Saturation improves the performance by using an efficient state encoding and a special search order. Other approaches, such as bounded model checking use different methods for improvement. During bounded model checking, the model is checked only up to a certain depth, limiting the size and complexity of the model checking problem.

The bounded and the saturation-based model checking use orthogonal ideas for improvements, therefore they could be combined. This may improve the saturation-based model checking techniques by reducing the run time in case of shallow requirements, where the bounded model checking typically excels. Although the high-level ideas of these two approaches are independent, in reality they affect each other. For example, the special search order makes limiting the exploration depth more difficult.

Goal. The goal of the work discussed in this chapter is to combine the principles of bounded model checking with the saturation-based techniques, to show its feasibility and to improve the performance of saturation; then to build an iterative bounded saturation-based CTL model checking algorithm and to assess its performance.

This chapter presents *B-I-Sat* (Bounded Iterative Saturation), an iterative, bounded CTL model checking algorithm using saturation-based techniques. Three different strategies are proposed, which are implementing the B-I-Sat algorithm with minor differences. The termination conditions, a crucial part of this algorithm, are analysed in more detail. The evaluation part shows that these methods improve the performance and scalability of saturation in certain cases, and may improve the applicability of model checking in industrial use cases too.

Structure of this chapter. Section 2.1 overviews the background of the work described in this chapter. The corresponding formal definitions can be found in Appendix A. Next, Section 2.2 is dedicated to the related work, with a special focus on the various saturation-based techniques. Section 2.3 discusses B-I-Sat, the novel iterative bounded model checking solution and its challenges. Two strategies, the restarting and continuing strategies are described here as well. The more advanced, so-called

compacting strategy is proposed in Section 2.4. After, Section 2.5 discusses the application of three-valued logic to give termination conditions for the iterative B-I-Sat algorithm. The evaluation of the proposed methods can be found in Section 2.6. Finally, Section 2.7 concludes the chapter.

2.1 Preliminaries

In this section we focus on the key preliminaries that are necessary to understand the contributions presented later. For a more detailed overview we refer to [*j*2; *a*30].

Model checking. *Model checking (see Def. A.1, p. 123)* [Cla08] determines whether a given model satisfies the given temporal logic formula (requirement). In general, model checking looks for the state set of a model M that satisfies the given formula f, but often we are interested only whether the initial state s_0 of the model satisfies f, formally: $M, s_0 \stackrel{?}{\models} f$ (or simply denoted as $s_0 \stackrel{?}{\models} f$).

Although the formal definition of model checking typically states that the model is given as a *Kripke structure (see Def. A.2, p. 123)*, modelling real systems using the low-level Kripke structures is often impractical due to the high number of states. Various higher-level models exist with different properties. The Petri net formalism *(see Def. A.4, p. 124)* is one of the widely used modelling methods, which will be used in this chapter as well. An example Petri net is shown in Figure 2.1.

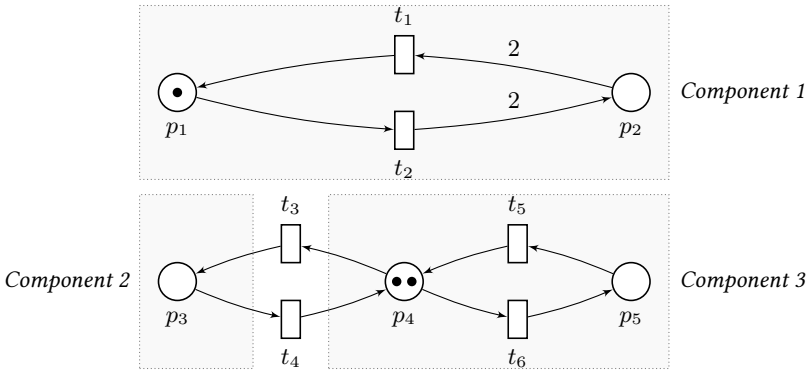


Figure 2.1: Example Petri net

Multiple formalisms exist to define the temporal logic formula too. Two well-known and widely-used formalisms are *computation tree logic* (CTL) and *linear temporal logic* (LTL). As saturation is mainly suitable for CTL [CMS03], we are focusing on this formalism. CTL expressions are interpreted on the *computation tree (see Def. A.5, p. 124)*, which is practically a radix tree of all possible paths from the initial state(s). CTL formulae *(see Def. A.6, p. 124)* [CE82] consist of Boolean expressions that may or may not be true for the individual states, and temporal operators (e.g. EF, EG, AX) which argue about where in the computation tree should a certain Boolean expression be true to satisfy the CTL formula. The intuitive meaning of the CTL operators can be seen in Figure 2.2 where for each formula a satisfying example computation tree is given.

Bounded model checking. *Bounded model checking* was introduced in [Bie+99] as a method based on SAT (Boolean satisfiability problem) solvers looking for counterexamples with iteratively increasing lengths. This idea can be generalised: bounded model checking evaluates the given temporal logic

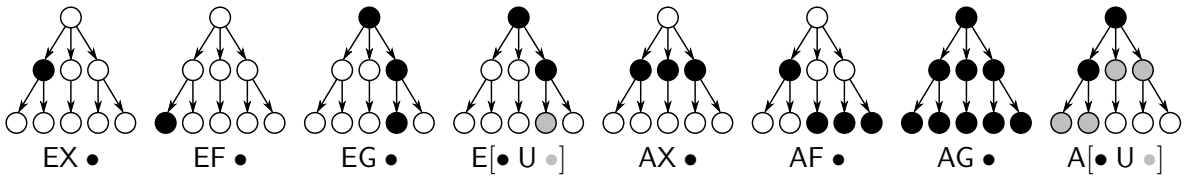


Figure 2.2: Examples of CTL operators

formula by iteratively checking greater and greater part of the model until the satisfaction of the requirement can be decided *(see Def. A.7, p. 124)*.

2.2 Related Work

This section overviews the work related to this chapter. As the goal of this work is to improve the saturation-based techniques, the focus is set to the various saturation algorithms proposed in the past.

2.2.1 Bounded Model Checking

Early symbolic model checking methods based on binary decision diagrams (BDDs) improved the scalability of model checking, making possible to use this verification method on industrial examples [Bur+92]. Although this pushed the limits of model checking compared to the explicit model checking algorithms, the memory bottleneck was still severe. Bounded model checking using SAT solver was proposed in [Bie+99] as a complementary technique. SAT-based model checking was widely considered more applicable than BDD-based methods [Bie+03; Aml+05]. Some authors have stated explicitly that BDD-based model checking is unable to "handle large state spaces of 'real world' designs" [Cha+02].

Bounded model checking is widely considered as a purely SAT-based method (e.g. [Aml+05]), however its principles can be implemented using other technologies too.

2.2.2 Saturation-Based Techniques

Saturation was first introduced in [CLS01] as a symbolic state space exploration method. Typically, the set of reachable states (the state space) is not explicitly given (enumerated) by a formal model, it has to be determined based on the initial state(s) and the next-state function. The simplest solution for that is a breadth- or depth-first search (BFS or DFS). Their time complexity is $O(|\mathcal{S}|)$, where $|\mathcal{S}|$ is the number of reachable states. In case of asynchronous systems, where many components are loosely coupled, this is not efficient. Imagine a system with two independent components, having n and m reachable states respectively. It is intuitive that the $O(nm)$ complexity of BFS could be reduced to $O(n+m)$ due to the independence.

This is the high-level motivation of saturation for checking state spaces of components in isolation as much as possible [CMS03]. Given a discrete-state model decomposed into K (not necessarily independent) parts (components), saturation determines the set of reachable states $\mathcal{S}$. Formally, the input model is as follows.

> **Definition 2.1 (Input model of saturation [CS03]).**
> A discrete-state model is a 4-tuple $M = \langle \hat{\mathcal{S}}, \mathcal{S}_0, \mathcal{E}, \mathcal{N} \rangle$, where:

- $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is the potential state space, decomposed into K parts ($\mathcal{S}_i$ is the local state space of component i);
- $\mathcal{S}_0 \subseteq \hat{\mathcal{S}}$ is the set of initial states;
- $\mathcal{E}$ is the set of events; and
- $\mathcal{N} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ is the next-state relation. It can also be regarded as a function $\mathcal{N}\colon \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$, where $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}'\colon (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$. ▪

The local state spaces $S_1, \ldots, S_K$ do not have to be explicitly given, as they can be explored on-the-fly [CMS03]. Furthermore, typically the $\mathcal{N}$ next-state relation is not given explicitly either, but based on the given high-level model $\mathcal{N}(\mathbf{s})$ can be determined for a given $\mathbf{s}$ (e.g. a Petri net implicitly defines $\mathcal{N}$). Notice that as the global state space is decomposed, each *global state* $\mathbf{s} \in \mathcal{S}$ is a composition of *local states*: $\mathbf{s} = (s_K, s_{K-1}, \ldots, s_1)$, where each $s_i \in \mathcal{S}_i$.

Underlying data structures. Saturation uses multivalued decision diagrams (MDDs) *(see Def. A.8, p. 125)* to store both the reachable state space $\mathcal{S}$ and the next-state relation $\mathcal{N}$. An MDD can efficiently encode a function $D_K \times D_{K-1} \times \cdots \times D_1 \to \{0, 1\}$, where each D_i is a finite domain. An MDD is a directed acyclic graph with a single root, where the edges are labelled with values from the D_i sets and the exactly two leaves of the graph are labelled by values 0 and 1 (so-called *terminal nodes*, denoted by $\mathbf{0}$ and $\mathbf{1}$). A path from the root to the terminal node $\mathbf{1}$ (or $\mathbf{0}$) with edge labels $x_K, x_{K-1}, \ldots, x_1$ encodes the mapping $(x_K, x_{K-1}, \ldots, x_1) \mapsto 1$ (or $(x_K, x_{K-1}, \ldots, x_1) \mapsto 0$). MDDs can also be used to store sets of K-tuples: a tuple $(x_K, x_{K-1}, \ldots, x_1)$ is an element of the set encoded by the MDD iff the MDD maps this tuple to 1. The set operations, e.g. union and intersection can also be applied to MDDs.

Example. *An example MDD can be seen in Figure 2.3(a) that encodes the following set (in x_3, x_2, x_1 order):*

$$\mathcal{S} = \{\ \begin{array}{l}(0,0,0), (3,0,0), (7,0,0), (2,1,0), (5,1,0), (4,2,0),\\ (0,0,1), (3,0,1), (7,0,1), (2,1,1), (5,1,1), (4,2,1)\end{array}\ \}.$$

Let us assume the following local state space encodings for the example Petri net in Figure 2.1.

$$\begin{array}{lll}
\textit{Component 3:} & \{ & 0 \mapsto \langle M(p_4) = 2, M(p_5) = 0\rangle, \quad 1 \mapsto \langle M(p_4) = 3, M(p_5) = 0\rangle,\\
& & 2 \mapsto \langle M(p_4) = 1, M(p_5) = 0\rangle, \quad 3 \mapsto \langle M(p_4) = 1, M(p_5) = 1\rangle,\\
& & 4 \mapsto \langle M(p_4) = 0, M(p_5) = 0\rangle, \quad 5 \mapsto \langle M(p_4) = 0, M(p_5) = 1\rangle,\\
& & 6 \mapsto \langle M(p_4) = 2, M(p_5) = 1\rangle, \quad 7 \mapsto \langle M(p_4) = 0, M(p_5) = 2\rangle \quad \}\\
\textit{Component 2:} & \{ & 0 \mapsto \langle M(p_3) = 0\rangle, \quad 1 \mapsto \langle M(p_3) = 1\rangle,\\
& & 2 \mapsto \langle M(p_3) = 2\rangle \quad \}\\
\textit{Component 1:} & \{ & 0 \mapsto \langle M(p_1) = 1, M(p_2) = 0\rangle, \quad 1 \mapsto \langle M(p_1) = 0, M(p_2) = 2\rangle \quad \}
\end{array}$$

Given these symbolic state encodings, the MDD in Figure 2.3(a) encodes the reachable state set of the Petri net in Figure 2.1.

The local state encodings in the above example were generated by the saturation algorithm. One can see that some local states are impossible to reach, e.g. state 1 of component 3. This is a side-effect of the on-the-fly local state space exploration. When a new potential local state is found, it is called *unconfirmed*. A local state that is later found to be globally reachable (i.e. it is part of a reachable global state) is called *confirmed*.

CTL model checking with saturation. As mentioned above, saturation was first introduced as a state space exploration algorithm. The state space exploration is practically the computation of the fixed point $\mathcal{S}_0 \cup \mathcal{N}(\mathcal{S}_0) \cup \mathcal{N}^2(\mathcal{S}_0) \cup \cdots = \mathcal{N}^*(\mathcal{S}_0)$ (where * denotes the reflexive and transitive closure [CMS06]). Based on this, saturation was later generalised as a fixed point computation algorithm and used for instance for CTL model checking, introduced first in [CS03]. CTL typically defines eight temporal operators (EX, EF, EG, EU, AX, AF, AG, AU), but it is enough to implement a (not necessarily minimal) generator subset of them. For example, each A operator can be expressed using E operators, therefore [CS03] proposes implementations only for the E operators.

The implementation of EX and EF operators is simple. EX can be formalised as follows: $\mathbf{s} \vDash \mathsf{EX}\, p$ iff $\exists \mathbf{s}' \in \mathcal{N}(\mathbf{s}) \colon \mathbf{s}' \vDash p$. Therefore if the set P of reachable states satisfying p ($P \subseteq \mathcal{S}$) is known, checking $\mathbf{s} \vDash \mathsf{EX}\, p$ is equivalent to checking $\mathbf{s} \overset{?}{\in} \mathcal{N}^{-1}(P)$. With similar reasoning, checking $\mathbf{s} \vDash \mathsf{EF}\, p$ (i.e. whether a state satisfying p is reachable from $\mathbf{s}$) can be reduced to checking $\mathbf{s} \overset{?}{\in} (\mathcal{N}^{-1})^*(P)$, which is a similar problem to determining the reachable state space of the model (i.e. $\mathcal{N}^*(\mathcal{S}_0)$). The implementation of EG and EU operators are less straightforward in symbolic settings and for the details we refer the reader to [CS03][1].

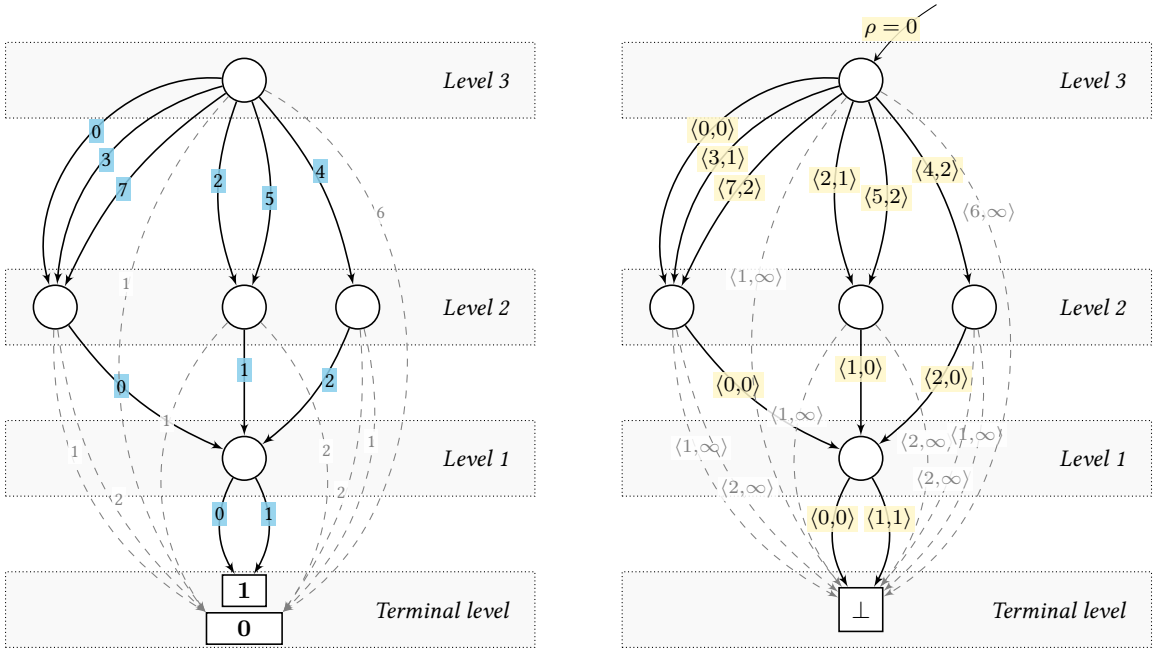
Constrained saturation. Constrained saturation was proposed by Zhao *et al.* in [ZC09] as an improvement for the CTL model checking. It limits the exploration to a given state set. While this can easily be achieved using the intersection MDD operation, MDD operations are typically expensive. Instead of the "step-and-cut" strategy (with intersections after each step), they have proposed a "check-and-step" strategy, where the constraint state set, encoded also by MDD, is simultaneously traversed and used for constraining the exploration. The implementation of EU operator with constrained saturation proved to be more efficient than the traditional approach. The formal details of constrained saturation can be found in [ZC09].

Bounded state space exploration with saturation. Saturation can efficiently store the reachable state space, but this does not mean that it can cope with every problem with reasonable amount of resources. Bounded model checking techniques are powerful when the requirement can be evaluated based on a fraction of the complete state space, for example to find shallow error states. However, the "non-standard search strategy" that makes saturation powerful in state space exploration makes its application for bounded state space exploration difficult [YCL09].

To perform saturation-based bounded exploration, the state distances (i.e. the minimum number of transitions to be fired to reach the state, see Def. A.3) have to be stored. Yu *et al.* [YCL09] considered different variants of decision diagrams. In the following, we will use their approach building on edge-valued MDDs (EV$^+$MDDs or EDDs) *(see Def. A.9, p. 125)*, which were also used in [CS02]. EDDs encode functions $D_K \times \cdots \times D_1 \to \mathbb{N} \cup \{\infty\}$. They can be regarded as extended MDDs, where each edge is additionally labelled with a non-negative integer (or ∞). Contrarily to MDDs, it is sufficient to have one leaf node in EDDs, the node $\perp$. A path from the root node to the leaf node $\perp$ maps the sum of the edge labels to the tuple encoded by the path.

Example. *An example EDD can be seen in Figure 2.3(b) that encodes the following mappings:*

[1] A saturation-based method was given for EG in [CS03], but even the authors of the paper consider its performance typically worse than the traditional greatest fixed point algorithm. The saturation-based implementation of EU was much better, but it was later significantly improved further in [ZC09].



(a) Example MDD

(b) Example EDD

Figure 2.3: Example decision diagrams (describing the symbolic state space of the Petri net shown in Figure 2.1)

$$\{\quad (0,0,0)\mapsto 0, (3,0,0)\mapsto 1, (7,0,0)\mapsto 2, (2,1,0)\mapsto 1, (5,1,0)\mapsto 2, (4,2,0)\mapsto 2,\\ (0,0,1)\mapsto 1, (3,0,1)\mapsto 2, (7,0,1)\mapsto 3, (2,1,1)\mapsto 2, (5,1,1)\mapsto 3, (4,2,1)\mapsto 3 \quad \}.$$

Given the state encodings discussed in the previous example, the EDD in Figure 2.3(b) encodes the reachable state set of the Petri net in Figure 2.1 with distance information (i.e. the EDD maps to each symbolic state its distance from the initial state $(0,0,0)$*).*

The calculated and stored distance information is then used to limit the state space exploration to the states with distance values less than or equal to the given bound. In case of BFS, it is easy to terminate the exploration when the bound is reached. However, the advantage of saturation makes this more challenging too. To overcome this challenge, Yu *et al.* added an explicit pruning after each step. Two pruning strategies were proposed: an *exact* method that strictly computes the bounded state space with respect to the given bound b, and an *approximative* method that is faster, but provides weaker guarantees: it only ensures that each state within bound b will be included and each state with distance more than $K \cdot b$ will be excluded from the state space (where K is the number of components in the model).

As the saturation-based techniques will be used for B-I-Sat as building blocks, deeper details are not necessary at this point. For more details we refer to [CLS01; CMS03; CZJ12]. The parts related to the contributions presented here are discussed in [*a*30], along with the pseudocode of the contributions. An overview and short summary of the main saturation-related papers is in Table 2.1.

Table 2.1: Milestones of the saturation algorithm

Ref.	Year	Contents, new contributions
[CLS00]	2000	First discussion of the main saturation concepts
[CLS01]	2001	The first paper drawing up the idea of *saturation*, a novel, MDD-based state space exploration algorithm
[CS02]	2002	State space exploration with state distances to generate shortest traces
[CMS03]	2003	Extension of saturation with automated explicit *local state space* discovery
[CS03]	2003	*CTL model checking* based on the saturation algorithm
[CY05]	2005	Conjunctive and disjunctive *partitioning*
[ZC09]	2009	*Constrained saturation* to improve the efficiency of CTL model checking
[YCL09]	2009	*Bounded reachability* checking based on saturation
[*c*28]	2012	Extension of saturation to *coloured Petri net* models
[*j*26]	2014	Introduction of the *lazy coloured saturation*
[CMS06]	2006	Summary and deep analysis of the previous saturation algorithms
[CZJ12]	2012	Summary of the first ten years of saturation

2.2.3 Bounded Model Checking With Decision Diagrams

Combining different model checking approaches is not unheard-of in the field of formal verification. For example, Chauhan *et al.* [Cha+02] combine SAT-based and BDD-based methods, where BDD-based model checking is only used on abstract models. The first work truly combining bounded model checking and decision diagram-based methods was done by Copty *et al.* [Cop+01] in 2001. Later, Cabodi *et al.* [CNQ05] used BDD-based bounded model checking and they have found that their approach can "deal with larger problems than other BDD-based tools" and that their "methodology seems to be more scalable with deeper bugs" than SAT-based bounded model checking methods [CNQ05].

The author is not aware of any work on combining bounded model checking principles with saturation-based techniques before the own work [*c*18].

2.3 Overview of the B-I-Sat Algorithm

This section presents the collected building blocks which can be used for a bounded saturation-based model checking algorithm. Then the high-level workflow of *B-I-Sat* (Bounded Iterative Saturation) is discussed. B-I-Sat is a novel CTL model checking algorithm that combines bounded model checking and saturation-based techniques. Finally, the arisen challenges and design questions are discussed.

2.3.1 Building Blocks

Here we review the main saturation-based techniques to collect the already existing building blocks that can be reused for the B-I-Sat algorithm.

Unbounded saturation-based model checking. The classic saturation-based CTL model checking [CS03] is a two-step procedure: first it explores the reachable state space, then the given CTL formula is evaluated, as can be seen in Figure 2.4.

It is easy to identify the following two building blocks.

State space exploration: It collects all reachable states in a given model from a given initial state.
- *Input:* Petri net (with initial marking and decomposition defined);
- *Outputs:* reachable state space $\mathcal{S}$ (MDD); next-state function $\mathcal{N}$ (MDD);
- *Internal results:* local state spaces $\mathcal{S}_k$; caches for the saturation functions and the decision diagram operations.

CTL formula evaluation: It evaluates the satisfaction of a given CTL formula on the given explored model.
- *Inputs:* state space $\mathcal{S}$ (MDD); initial state(s) $\mathcal{S}_0$ (MDD); next-state function $\mathcal{N}$ (MDD); CTL formula f;
- *Output:* Boolean result (true iff f is satisfied on $\mathcal{S}$ by $\mathcal{S}_0$) and optionally the subset of $\mathcal{S}$ which satisfies f;
- *Internal results:* caches for the saturation functions and the decision diagram operations.

Bounded state space exploration. The bounded state space exploration [ZC09] is a single-step process, it will be considered as a single building block. Let us introduce a new notation: $\mathcal{S}_{[a;b]} \triangleq \{\mathbf{s} \in \mathcal{S} \colon a \leq \delta(\mathbf{s}) \leq b\}$, i.e. $\mathcal{S}_{[a;b]}$ is the partial state space containing all states with distances between a and b (both inclusive).

Bounded state space exploration (with exact truncation strategy): It collects all reachable states from a given initial state in a given model which are within a given distance b.
- *Input:* Petri net (with initial marking and decomposition defined); bound b;
- *Output:* reachable (partial or bounded) state space $\mathcal{S}_{[0;b]}$ (EDD); (partial) next-state function $\mathcal{N}$ (MDD);
- *Internal result:* local state spaces $\mathcal{S}_k$; caches for the saturation functions and the decision diagram operations.

Note that only the exact truncation (pruning) strategy ensures that the resulting state space is $\mathcal{S}_{[0;b]}$. The approximative truncation ensures only that the resulting state space $\mathcal{S}'$ contains all states within bound b and contains no states with distance more than $K \cdot b$: $\mathcal{S}_{[0;b]} \subseteq \mathcal{S}' \subseteq \mathcal{S}_{[0;K \cdot b]}$.

2.3.2 Sketching Up the B-I-Sat Algorithm

To achieve the goal and to design a bounded saturation-based CTL model checking algorithm, two key challenges should be met. First, the unbounded saturation-based algorithm should work on a partial (bounded) state space, using the bounded state space exploration algorithm. Second, the algorithm should be made iterative, to check the CTL formula on an increasingly large part of the full state space.

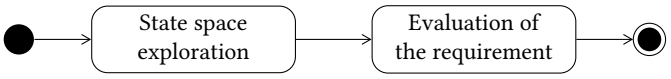


Figure 2.4: Overview of the classic saturation-based model checking

Non-iterative bounded model checking with saturation. The apparent challenge of integrating the bounded state space exploration to the saturation-based CTL model checking based on the discussion in Section 2.3.1 is the mismatch between the state space encodings: the explored partial state space is encoded by an EDD, while the CTL evaluation works on an MDD state space. The CTL evaluation could be modified to handle EDDs, but this might have a negative impact on the performance, as the MDD representation of the state space is typically more compact than the EDD representation. The basic CTL evaluation algorithm cannot benefit from the distance information, therefore they can be dropped from the EDD. An MDD which encodes a state set contains a state $\mathbf{s}$ if the corresponding path leads to the node $\mathbf{1}$. In an EDD which encodes a state set, a state $\mathbf{s}$ is contained if the corresponding path leading to the terminal node has a finite weight. Therefore if the EDD encodes a function f_E, the corresponding MDD should encode the function f_M as follows [*a*31]:

$$f_M(x_n, \ldots, x_1) = \begin{cases} 0, \text{ if } f_E(x_n, \ldots, x_1) = \infty \\ 1, \text{ if } f_E(x_n, \ldots, x_1) < \infty \end{cases} .$$

This can be used to construct an MDD corresponding to the state set encoded by the EDD for the CTL formula evaluation. This transformation has a linear time complexity in the number of EDD nodes. The MDD representing the set encoded by the EDD $\mathcal{E}$ without the distance information will be denoted by $MDD(\mathcal{E})$. Similarly, let us denote by $EDD(\mathcal{M})$ the EDD which maps the value 0 exactly to the tuples contained by the MDD $\mathcal{M}$, and maps ∞ to all the other tuples.

The second problem is less apparent and it is related to the pruning during bounded state space exploration. The pruning step will remove the states that are not within the given bound, but this does not modify the next-state relation. Therefore the CTL formula evaluation may consider states that are not included in the bounded state space. This is not a threat to correctness, but it reduces the advantage of bounded model checking [*e*21; *c*17]. This challenge will be discussed later in detail.

If these challenges are solved, a simple, non-iterative, saturation-based bounded model checking algorithm can be drawn up, as shown in Figure 2.5. Obviously, this is not useful in practice yet, as in cases where the bound is not chosen correctly, the result will not be representative of the whole state space. Knowing the smallest necessary bound value a priori is typically not possible.

Figure 2.5: Overview of the non-iterative, saturation-based bounded model checking

Iterative bounded model checking with saturation. The need to know the correct bound value can be eliminated by making the model checking method presented in Figure 2.5 iterative. Looking at a high level this is simple: the state space should be explored up to an increasing bound b, until a representative result is available and the procedure can be terminated. This is the main idea of the B-I-Sat algorithm, depicted in Figure 2.6. However, if we take a closer look at different parts of the algorithm, various challenges and questions arise. They are discussed in the next section.

2.3.3 Challenges and Solutions

Previously the high-level ideas of B-I-Sat were presented without details. In the following part of this section the main design questions, challenges and the given solutions are summarised.

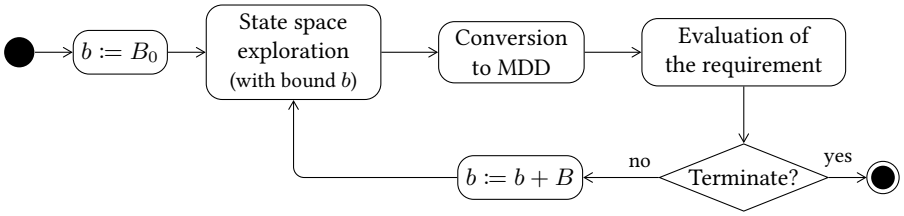


Figure 2.6: Overview of the iterative, saturation-based bounded model checking (B-I-Sat)

- **Initial bound and bound increment.** Most SAT-based bounded model checking algorithms start with bound 0 and increment it by one in every iteration. This ensures to find the shortest counterexample and to explore the smallest necessary part of the state space. In case of saturation this might not be the most efficient strategy. Therefore the initial bound B_0 and the bound increment B will not be fixed and can be used to parametrise the algorithm, depending on the current model and requirement. To simplify the discussion, in the following we will assume that $B_0 = B$, thus the bound b in iteration i is $b = i \cdot B$. However, all the presented strategies can be generalised to use initial bounds different from the increments ($B_0 \neq B$).
- **Pruning strategy.** Contrarily to BFS, due to the irregular search order, an explicit pruning operation has to be included in the saturation algorithm. Two pruning strategies were proposed in [YCL09]: an exact truncation and a faster but approximative truncation. The approximative truncation had a better performance in [YCL09], however it was shown in [c18] that the use of caches can make the exact truncation strategy competitive. The choice of pruning strategy will not be fixed. In the following, we will generally assume the use of the exact truncation strategy, as it simplifies the notations and the discussion in the following.
- **Termination.** After the first exploration and evaluation steps, a result is available. However, this result does not necessarily hold for the whole state space. If it does, the model checking algorithm can terminate. Otherwise, a next iteration with greater bound is required. Deciding when to terminate the algorithm is not simple. This question will be addressed in detail in Section 2.5, and termination conditions will be proposed based on three-valued logic.
- **Avoiding unreachable states during requirement evaluation.** As it was discussed before, the next-state relation $\mathcal{N}$ may contain transitions leaving states that are not included in the bounded state space. This implies that the CTL formula evaluation, which uses $\mathcal{N}^{-1}$, may include unreachable states in the resulting state sets. This is not a threat to correctness, but depending on the model it may significantly decrease the performance of the algorithm. A straightforward solution is to intersect the result in each iteration with the explored partial state space $\mathcal{S}_{[0,b]}$, but this might be an expensive operation. Similar problems were addressed in case of the evaluation of EU CTL operators in [ZC09]. The proposed solution, the constrained saturation restricts the exploration to a given set $\mathcal{C}$, without explicit intersection operation, resulting in a better performance. The same solution can be used for bounded CTL model checking as well: using constrained saturation, the exploration can be restricted to the explored partial state space $\mathcal{S}_{[0,b]}$. All proposed strategies will use constrained saturation for the CTL requirement evaluation part with the constraint $\mathcal{C} = \mathcal{S}_{[0,b]}$.
- **Reusing the produced data.** The state space exploration step results in various data: partial state space, next-state function, local state spaces, cache values, etc. Dropping all this data at the

beginning of each iteration and restarting the exploration without any knowledge about the previous iterations trivially results in a correct algorithm. However, keeping some of the data may improve the performance. In the following, various strategies will be discussed which mainly differ in this aspect, i.e. how do they reuse data from previous iterations. See Sections 2.3.4 and 2.4 for more details.

2.3.4 Iteration Strategies

In this section two simple iteration strategies are proposed for B-I-Sat: the *restarting* and *continuing* strategies. The pseudocode of the presented iteration strategies can be found in Appendix B (p. 127).

Restarting strategy. The simplest iteration strategy would be to start each iteration with no prior information. This strategy would start every iteration from the initial state (set) $\mathcal{S}_0$, then explore the part $\mathcal{S}_{[0;i\cdot B]}$ in each iteration $i = 1, 2, \ldots$ (where B is the given bound increment).

Based on Section 2.3.1, there are four types of data produced by the state space exploration: reachable state spaces (encoded by EDDs), local state sets, next-state relations, and cache data.

- The state space EDD built in the last iteration will be dropped. This is the simplest way to deal with the fact that increasing the bound will make some of the nodes unsaturated, i.e. some nodes will not represent fixed points anymore with respect to the next-state relation (which grows if the bound increases).
- The next-state relation ($\mathcal{N}$) can grow only if a greater bound is used, therefore it can be reused and extended.
- The local state spaces ($\mathcal{S}_i$) can grow only if a greater bound is used, therefore the local state spaces from the previous iterations can be used as "hints" for the on-the-fly exploration algorithm and can improve the performance of saturation in iterations $i > 1$.
- The caches related to the state space EDD should be flushed, as their content is not valid after restarting the exploration. All other cache entries may be kept.

Figure 2.7 illustrates the restarting strategy. The yellow part (with a thick border) represents the initial state set used in each iteration. The grey parts symbolise the newly explored state sets. Algorithm B.1 (p. 128) shows the pseudocode of the restarting strategy. More details about the implementation can be found in [*a*30].

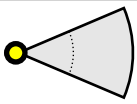
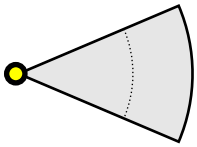
	State space ($\mathcal{S}'$)	**Part encoded by $\mathcal{S}'$**
Iteration 1		$[0; B]$
Iteration 2		$[0; 2 \cdot B]$
Iteration 3		$[0; 3 \cdot B]$

Figure 2.7: Illustration of the restarting strategy (based on [*a*30])

Here we assume that the exact truncation is used as pruning strategy. However, the restarting strategy can be used with the approximative truncation as well. In this case, obviously the partial state space explored in iteration i will be a superset of $\mathcal{S}_{[0;i\cdot B]}$.

Continuing strategy. At first look it seems to be wasteful to restart each iteration from the initial state of the model, as for every bound $b' > b : \mathcal{S}_{[0;b]} \subseteq \mathcal{S}_{[0;b']}$, i.e. every state $\mathbf{s}$ found in iteration i will be found in every future iteration, with the same distance value. This motivated the continuing strategy, which starts every iteration $i > 1$ from the result state space of iteration $i - 1$ as initial state set. The details of the minor technical modifications required on the saturation-based state space to support this are discussed in [*a*30].

It is important to notice that it is not known at the beginning of an iteration, which nodes are saturated (i.e. no new states can be found from them) and which are not, thus it is required to resaturate (i.e. to recompute the fixed points) every node. This may be an expensive operation, but it is not obvious whether this needs more or less resources compared to the restarting strategy. This question will be addressed in the evaluation section (see Section 2.6). In every other detail the continuing strategy works in the same way as the restarting strategy.

Figure 2.8 illustrates the continuing strategy. The yellow part (with a thick border) represents the initial state set used in each iteration. The grey parts symbolise the newly explored state sets. Algorithm B.2 (p. 128) shows the pseudocode of the continuing strategy. More details about the implementation can be found in [*a*30].

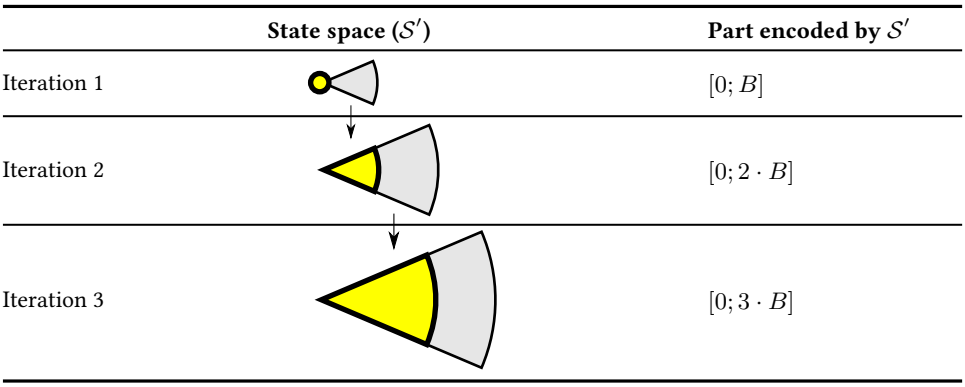


Figure 2.8: Illustration of the continuing strategy based on [*a*30])

Here we have assumed that the exact truncation was used as a pruning strategy, but just as in the case of the restarting strategy, the approximate truncation can be used as well.

The next section introduces a third, more complex strategy, the compacting strategy.

Publications related to this section. The first analysis of combining bounded state space exploration and saturation-based CTL model checking was presented in [*c*18; *a*32], then more details were discussed in [*a*31; *a*30]. The presented discussion of the challenges and the given solutions are based on [*a*30]. The idea to improve the algorithm by using constrained saturation was described in [*e*21] and the details of the improved version were presented in [*c*17]. The restarting strategy was first presented in [*c*18], the continuing strategy in [*c*17].

2.4 Compacting Saturation Strategy[2]

Both the restarting and the continuing strategies have a potential weakness: they store the whole $[0; b]$ part of the state space (i.e. $\mathcal{S}_{[0;b]}$, the states with distances between 0 and b) in a single EDD. The algorithms use EDDs for storing the distance information from the initial state along with each state, which is not possible in case of the simple MDD encoding. This information is necessary to limit the state space exploration at the bound b. The storage of the distance has its price to pay: the EDD representation of the state space is typically less compact than the MDD representation (without the distance information) would be.

While it is unavoidable to store distance information for some of the explored states to limit the exploration at the given bound b, it is not necessary to store it for every state. Consider the nth iteration ($n > 1$). In this case, the $\mathcal{S}_{[0;n\cdot B]}$ should be the result of the bounded state space exploration. However, the $\mathcal{S}_{[0;(n-1)\cdot B]}$ part was already explored in the previous iteration, and it is known that these states will be present in $\mathcal{S}_{[0;n\cdot B]}$ too. These states could be stored in the more compact MDD format. The main idea of the *compacting saturation* strategy is to store the state spaces discovered in previous iterations using a more compact, MDD-based representation.

Initial state set. In case of restarting and continuing strategies, each iteration was started either from the initial state or from the state space of the previous iteration. The goal of compacting saturation is to avoid the EDD encoding for states discovered during previous iterations, if possible. Therefore this exploration cannot be started from the initial state or from the previous iteration's state space, as it would require the EDD encoding of these states.

It is easy to see that starting the iteration $n > 1$ from $\mathcal{S}_{[0;(n-1)\cdot B]}$ is unnecessary: if every state with distance at most $(n-1)\cdot B$ is already explored, new states can only be found from states with distances *exactly* $(n-1)\cdot B$. Therefore new states can only be found from the states at the "frontier of the bounded state space", i.e. from the states in $\mathcal{S}_{[(n-1)\cdot B;(n-1)\cdot B]}$, the set containing states $\mathbf{s}$ with $\delta(\mathbf{s}) = (n-1)\cdot B$. This frontier state set will be the initial state set of the iteration n.

Details of the compacting strategy. In the following, the steps of the new compacting saturation algorithm are described in detail. Let us denote the initial state set of iteration n by $\mathcal{I}_n$. The result state space encoded by an EDD will be $\mathcal{S}_{[(n-1)\cdot B;n\cdot B]}$ after iteration n. The frontier of this state space is denoted by $\mathcal{F}_n = \mathcal{S}_{[n\cdot B;n\cdot B]}$. The state space explored in the first n iterations without distance information (i.e. represented by an MDD) is $\mathcal{M}_n$.

The *first iteration* of the compacting saturation is started from the state (set) $\mathcal{I}_1 = EDD(\mathcal{S}_0)$ and it explores the state space until the bound $b = B$. The result is the state set $\mathcal{S}_{[0;B]}$ encoded by EDD. Then the algorithm computes the frontier of the state space, i.e. $\mathcal{S}_{[B;B]} = \mathcal{F}_1$ and converts the EDD of $\mathcal{S}_{[0;B]}$ to an MDD representation $\mathcal{M}_1 = \mathcal{M}'_1$. The CTL formula will be evaluated on $\mathcal{M}_1$.

After, in *each iteration* $n > 1$ where the current bound is $b = n\cdot B$, the exploration is started from $\mathcal{I}_n = \mathcal{F}_{n-1} = \mathcal{S}_{[(n-1)\cdot B;(n-1)\cdot B]}$. The state space is explored until the bound b that is $\mathcal{S}_{[(n-1)\cdot B;n\cdot B]} = \mathcal{S}_{[b-B;b]}$. Next, the algorithm computes the frontier of the state space $\mathcal{F}_n = \mathcal{S}_{[b;b]}$ and converts the EDD of the state space explored in the current iteration ($\mathcal{S}_{[b-B;b]}$) to an MDD representation $\mathcal{M}'_n$. The evaluation of the CTL formula will be performed on the MDD $\mathcal{M}_n = \mathcal{M}'_n \cup \mathcal{M}_{n-1} = \bigcup_{i=1}^{n} \mathcal{M}'_i$. A summary of these data structures in the different iterations can be seen in Table 2.2.

[2]This section is an extended and adapted version of Section 4.2 of [j2].

Table 2.2: Overview of the different data structures of the compacting strategy

Iteration (i)	**Initial states** ($\mathcal{I}$ EDD)	**Explored states** ($\mathcal{S}$ EDD)	**Frontier set** ($\mathcal{F}$ EDD)	**State set for CTL evaluation** ($\mathcal{M}$ MDD)
1	$EDD(\mathcal{S}_0)$	$\mathcal{S}_{[0;B]}$	$\mathcal{F}_1 = \mathcal{S}_{[B;B]}$	$\mathcal{M}_1 = MDD(\mathcal{S}_{[0;B]})$
2	$\mathcal{F}_1$	$\mathcal{S}_{[B;2B]}$	$\mathcal{F}_2 = \mathcal{S}_{[2B;2B]}$	$\mathcal{M}_2 = \mathcal{M}_1 \cup MDD(\mathcal{S}_{[B;2B]})$
3	$\mathcal{F}_2$	$\mathcal{S}_{[2B;3B]}$	$\mathcal{F}_3 = \mathcal{S}_{[3B;3B]}$	$\mathcal{M}_3 = \mathcal{M}_2 \cup MDD(\mathcal{S}_{[2B;3B]})$
…	…	…	…	…

The frontier sets $\mathcal{F}_i$ can be computed using a modified version of the exact truncation operation described in [YCL09], as introduced in [*a*30]. The pseudocode of the algorithm is shown in Algorithm B.4 (p. 130).

Avoiding to revisit states. During the traversal, the previously explored states ($\mathcal{M}$) should not be re-explored: no states of $\mathcal{M}_n = \bigcup_{i=1}^{n} \mathcal{M}'_i$ should be explored again in iteration $n+1$. To be more precise, the goal of the algorithm is to keep: $\mathcal{M}'_{n+1} \cap (\mathcal{M}_n \setminus \mathcal{I}_{n+1}) = \emptyset$ (overlap in the initial state set is allowed). This is an obvious consequence of the state space parts defined above, i.e.

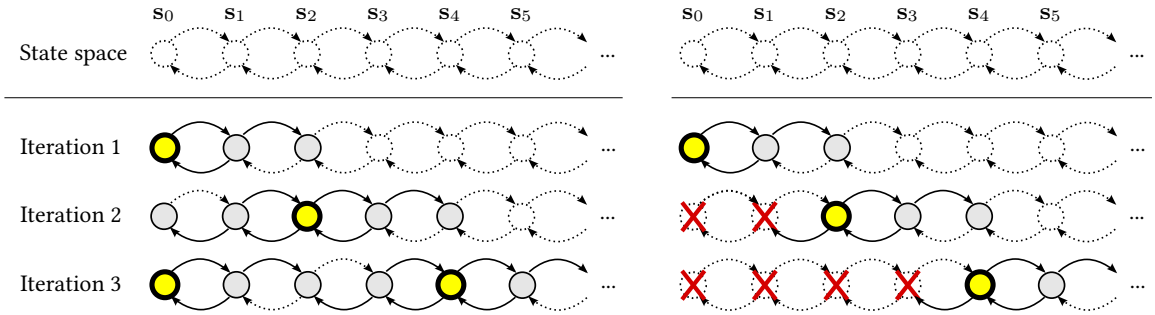
$$\mathcal{S}_{[n\cdot B;(n+1)\cdot B]} \cap \mathcal{S}_{[(n+1)\cdot B;(n+2)\cdot B]} = \mathcal{S}_{[(n+1)\cdot B;(n+1)\cdot B]} = \mathcal{F}_{n+1}.$$

However, if a state $\mathbf{s} \in \mathcal{M}_{n-1}$ is reachable from $\mathcal{I}_n$, it can be explored during the iteration n causing that it will be part of $\mathcal{M}'_n$ too. It effectively means that the state $\mathbf{s}$ will be stored with two different distance values. The algorithm has to prevent this situation for efficiency reasons and also to keep the correctness. The following example illustrates this issue.

Example. *The problem of revisiting already explored states is illustrated in Figure 2.9. The example uses a simple state space with states $\mathbf{s}_0, \mathbf{s}_1, \ldots$, where for each $\mathbf{s}_i$ state the transitions to $\mathbf{s}_{i+1}$ and to $\mathbf{s}_{i-1}$ (if $i > 0$) are possible. In the examples the grey circles denote concrete states explored in the current iteration (with $B_0 = B = 2$), the yellow states (with a thick border) symbolise the initial states of the current iteration. Figure 2.9(a) shows that without explicitly avoiding to revisit the already explored states, many states will be visited several times, causing degraded performance and potentially incorrect distance values. If the previously visited states are explicitly excluded from the exploration (denoted with red crosses in Figure 2.9(b)), the set of explored states in each iteration is the correct one.*

A simple solution would be to subtract $\mathcal{M}_{n-1}$ from the state set of the iteration $n > 1$ after each step. It would make the method correct, but not efficient. The problem is similar to the motivation of the constrained saturation, where an intersection operation would be required after each step. The difference is that here the forbidden states should be excluded, i.e. a subtraction operation is needed. The same principles can be used and constrained saturation can be modified to respect the given set of forbidden states [*a*30]. We call this the *negated constrained saturation*. For the details and pseudocode of negated constrained saturation we refer the reader to [*a*30]. If we use the negated constrained saturation with $\mathcal{M}_{n-1}$ as set of forbidden states, the re-exploration of already explored states can be efficiently avoided.

The compacting strategy is illustrated by Figure 2.10. The yellow part (with a thick border) represents the initial state set used in each iteration. The grey parts symbolise the newly explored state sets. The dark blue parts are the states at the frontier of the state space. Algorithm B.3 (p. 129) shows the pseudocode of the compacting strategy. More details about the design and implementation can be found in [*a*30].



(a) Compacting saturation without excluding the already visited states

(b) Compacting saturation with excluding the already visited states

Figure 2.9: Example about avoiding revisiting already explored states

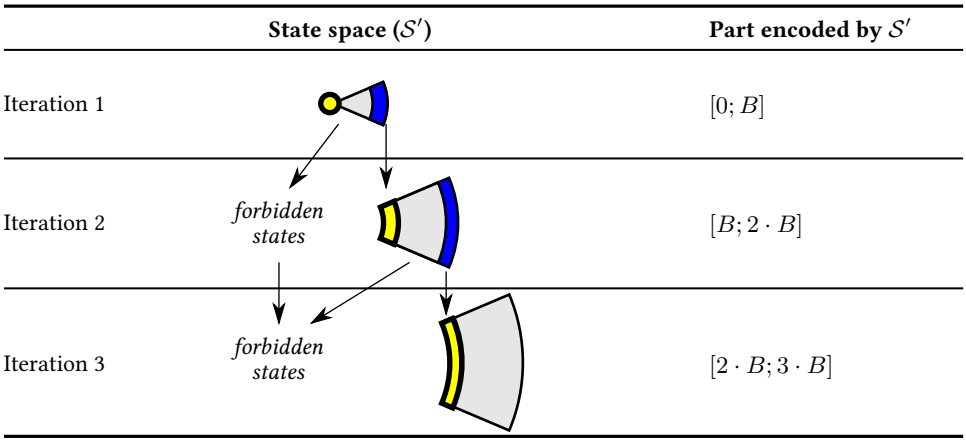


Figure 2.10: Illustration of the compacting strategy based on [*a*30])

Publications related to this section. The compacting strategy was first presented in [*a*30], then reformalised in [*j*2]. The three B-I-Sat iteration strategies were discussed and compared in [*a*30] and [*j*2].

2.5 Termination Conditions

Formally, the result of model checking *(see Def. A.1, p. 123)* is a state set S of a model M that satisfies the given formula f. Typically, we call a formula f *satisfied*, if the initial state (or one of the initial states) of the model is in the state set S, and f is *not satisfied* otherwise. Therefore the two-valued Boolean logic (with values $\mathbb{B} = \{T, F\} = \{1, 0\}$, standing for true and false values) can be used to argue about the satisfaction of a formula. In case of bounded model checking this is not necessarily true: if the current bound b is too small, it might not be known based on the partially explored state space whether the formula f is satisfied or not by the model M.

Three-valued logic [Kle52] introduces a third value $\perp$, representing the value *unknown*: $\mathbb{T} = \{T, F, \perp\}$. The common Boolean operators $\neg, \vee, \wedge$ can also be extended to use the new value $\perp$, see Figure 2.11. If the inclusion of any of the initial states in the set of states satisfying f is described using three-valued logic, it is easy to give a termination condition for the iterative B-I-Sat algorithm: the algorithm terminates iff for any initial state the inclusion is T or for all initial states the inclusion is

x	$\neg x$
$\bot$	$\bot$
F	T
T	F

(a) Truth table of negation

$\wedge$	$\bot$	F	T
$\bot$	$\bot$	F	$\bot$
F	F	F	F
T	$\bot$	F	T

(b) Truth table of $\wedge$ operation

$\vee$	$\bot$	F	T
$\bot$	$\bot$	$\bot$	T
F	$\bot$	F	T
T	T	T	T

(c) Truth table of $\vee$ operation

Figure 2.11: Truth tables of basic three-valued logic operators [SS05]

F. For Petri nets, where there is only one initial state this is simplified to the following: the algorithm terminates iff the inclusion of the initial state in the set of states satisfying f is not $\bot$. It has to be noted that three-valued logic in bounded model checking was already used in [BG99; SS05], but in B-I-Sat it is applied to CTL formulae.

2.5.1 Notations

Given a partial state space $\mathcal{S}^b$ that is the result of a bounded state space exploration in a certain iteration, let us denote by $\mathcal{S}^b_\phi$ the states in which ϕ is evaluated to true with respect to the state space $\mathcal{S}^b$ ($\mathcal{S}^b_\phi \subseteq \mathcal{S}^b$). Practically, this is the result given by the CTL evaluation algorithm. However, this result cannot be lifted to the full model, the current partial state space $\mathcal{S}^b$ may not contain enough information to decide whether ϕ is satisfied or not. For example, if ϕ checks the reachability of any state with label p ($\phi = \mathsf{EF}\ p$), the fact that ϕ does not hold for the initial state with a certain bound b does not imply that with a bound $b' > b$ the formula ϕ cannot be satisfied. If there are reachable states with label p, but they are all for a distance at least d, in every iteration with bound $b < d$ the result will be *not satisfied*, even though based on the full state space the requirement should be *satisfied*.

Let us denote by $\mathcal{S}_\phi$ the set of states where the CTL formula ϕ is true *based on the full model*. This set is not known before the full state space is explored, this is why we will use three-valued logic to argue about the result based on the partial state space. We introduce an evaluation function $e_\phi \colon \mathcal{S}^b \to \mathbb{T}$. This will determine for each known state whether it is in set $\mathcal{S}_\phi$ or not *based on the current, partial knowledge*. $e_\phi(\mathbf{s}) = T$ means that based on the partial state space explored it can be concluded that $\mathbf{s} \in \mathcal{S}_\phi$. If $e_\phi(\mathbf{s}) = \bot$, it means that based on the partially explored state space it cannot be decided whether $\mathbf{s} \in \mathcal{S}_\phi$ or not.

Now the termination condition can be formalised too. As the goal of the model checking is to decide whether the initial state $\mathbf{s}_0$ is in $\mathcal{S}_\phi$ or not (assuming one single initial state $\mathbf{s}_0$), the iterative model checking can be terminated iff $e_\phi(\mathbf{s}_0) \neq \bot$.

In the following, we discuss how the function e_ϕ can be constructed based on partial state spaces ($\mathcal{S}^b$) and evaluations on partial state spaces ($\mathcal{S}^b_\phi$). For the sake of simpler discussion we assume that the exact truncation is used as pruning strategy.

2.5.2 Evaluation of CTL Operators

Let us start with a trivial case, when the full state space is explored, i.e. $\mathcal{S} = \mathcal{S}^b$. In this case the evaluation is obvious for any formula:

$$e_\phi(\mathbf{s}) = \begin{cases} T & \text{if } \mathbf{s} \in \mathcal{S}^b_\phi, \\ F & \text{if } \mathbf{s} \notin \mathcal{S}^b_\phi \end{cases}.$$

The iterative bounded model checking can trivially be terminated if the full state space is explored, as there are no unknown states. Detecting if this is the case is simple if the exact truncation is used as pruning strategy: when the frontier state set is empty, no new states can be found. Detection of the exploration of the full state space (i.e. the situation when $\mathcal{S} = \mathcal{S}^b$) can be done in more efficient ways in the case of certain strategies, without computing the frontier set [*a*30]. The detection is more complex if the approximative pruning strategy is used. For the details we refer the reader to [*a*30].

For the rest of the section we assume that the full state space is not yet explored, i.e. there is at least one reachable state that is not in the partial state space $\mathcal{S}^b$.

In the following the three-valued evaluation of the various CTL temporal is discussed. This discussion is based on [*j*4; *a*31], but the formalisation is modified and adapted to the rest of this dissertation. Note that due to duality only E CTL operators will be considered in the following. Also, we assume that ϕ is a simple CTL expression, i.e. it is one CTL temporal operator pair with Boolean arguments. This excludes the expressions with nested CTL temporal operators (e.g. $\mathsf{EG}(\mathsf{EF}\ p)$).

Table 2.7 (p. 36) shows examples of the evaluation of the E CTL operators. In each case the grey circle represents $\mathcal{S}^b$. The states labelled with $*$ can be arbitrary, i.e. no matter if they satisfy p or not, the example holds.

EX operator. The EX CTL operator simply checks the existence of a successor state with a given property. If in a certain iteration the $\mathsf{EX}\ p$ formula is satisfied by the states in $\mathcal{S}^b_{\mathsf{EX}\ p}$, the $e_{\mathsf{EX}\ p}$ will be defined as follows.

- $e_{\mathsf{EX}\ p}(\mathbf{s}) = T$, if $\mathbf{s} \in \mathcal{S}^b_{\mathsf{EX}\ p}$.
- $e_{\mathsf{EX}\ p}(\mathbf{s}) = F$, if $\mathbf{s} \notin \mathcal{S}^b_{\mathsf{EX}\ p}$ and $\mathbf{s}$ is not on the frontier of the bounded state space, i.e. all successor states of $\mathbf{s}$ are included in the explored partial state space.
- $e_{\mathsf{EX}\ p}(\mathbf{s}) = \bot$, otherwise.

EF operator. The EF CTL operator computes a least fixed point, determining all states from which a state set satisfying a given property is reachable. If in a certain iteration the $\mathsf{EF}\ p$ formula is satisfied by the states in $\mathcal{S}^b_{\mathsf{EF}\ p}$, the $e_{\mathsf{EF}\ p}$ will be defined as follows.

- $e_{\mathsf{EF}\ p}(\mathbf{s}) = T$, if $\mathbf{s} \in \mathcal{S}^b_{\mathsf{EF}\ p}$ (i.e. a state satisfying p is reachable from $\mathbf{s}$ already in the partial state space).
- $e_{\mathsf{EF}\ p}(\mathbf{s}) = F$, if all successor states of $\mathbf{s}$ are explored (and none of them is on the frontier of the state space, i.e. it is not possible to find new successors). Notice that this case may only apply to the initial state $\mathbf{s}_0$ if the full state space is explored.
- $e_{\mathsf{EF}\ p}(\mathbf{s}) = \bot$, otherwise. If there are unexplored reachable states from $\mathbf{s}$ it cannot be decided whether any of them will satisfy p or not.

EU operator. The EU CTL operator computes a least fixed point, determining all states from which a state set satisfying a given property q is reachable through states satisfying p. If in a certain iteration the $\mathsf{E}[p\ \mathsf{U}\ q]$ formula is satisfied by the states in $\mathcal{S}^b_{\mathsf{E}[p\ \mathsf{U}\ q]}$, the $e_{\mathsf{E}[p\ \mathsf{U}\ q]}$ will be defined as follows.

- $e_{\mathsf{E}[p\ \mathsf{U}\ q]}(\mathbf{s}) = T$, if $\mathbf{s} \in \mathcal{S}^b_{\mathsf{E}[p\ \mathsf{U}\ q]}$ (i.e. a state satisfying q is reachable from $\mathbf{s}$ through states in p already in the partial state space).
- $e_{\mathsf{E}[p\ \mathsf{U}\ q]}(\mathbf{s}) = F$, if $\mathbf{s} \notin \mathcal{S}^b_{\mathsf{E}[p\ \mathsf{U}\ q]}$ and there is no path from $\mathbf{s}$ to the frontier of the explored state space whose states are all labelled with p. This means that no new explored state can result in a through-p-to-q path, the state $\mathbf{s}$ cannot satisfy the given requirement.
- $e_{\mathsf{E}[p\ \mathsf{U}\ q]}(\mathbf{s}) = \bot$, otherwise.

EG operator. A state $\mathbf{s}$ satisfies the $\mathsf{EG}\ p$ formula iff there exists one cycle in the state space graph where all states are labelled with p and this cycle is reachable from $\mathbf{s}$ through states labelled with p (there exists a so-called p-lasso). If in a certain iteration the $\mathsf{EG}\ p$ formula is satisfied by the states in $\mathcal{S}^b_{\mathsf{EG}\ p}$, the $e_{\mathsf{EG}\ p}$ will be defined as follows.

- $e_{\mathsf{EG}\ p}(\mathbf{s}) = T$, if $\mathbf{s} \in \mathcal{S}^b_{\mathsf{EG}\ p}$ (i.e. a p-lasso is reachable from $\mathbf{s}$ already in the partial state space, thus it is necessarily reachable in the full state space too).
- $e_{\mathsf{EG}\ p}(\mathbf{s}) = F$, if $\mathbf{s} \notin \mathcal{S}^b_{\mathsf{EG}\ p}$ and there is no path from $\mathbf{s}$ to the frontier of the explored state space whose states are all labelled with p. This means that no new explored state can result in a p-lasso, the state $\mathbf{s}$ cannot satisfy the given requirement.
- $e_{\mathsf{EG}\ p}(\mathbf{s}) = \bot$, otherwise.

Both for EG and EU operators it might be necessary to determine if there exists a path from state $\mathbf{s}$ to the frontier of the bounded state space, where each state satisfies p. This can be determined by evaluating another CTL formula. Let $\mathcal{F} \subset \mathcal{S}^b$ represent the set of states which are at the frontier of the currently explored state space. Determining the frontier set is discussed in Section 2.4. Then the CTL formula $\mathsf{E}[p\ \mathsf{U}\ (\mathcal{F} \cap p)]$ holds[3] for a state $\mathbf{s}$ in the partial state space iff it has at least one path to the frontier of the state space where all states are labelled with p.

Complex CTL expressions. In the discussion above it was assumed that the given formula ϕ is a simple CTL expression. This imposed two restrictions: ϕ does not contain CTL expressions connected with Boolean operators (e.g. $\mathsf{EF}\ p \wedge \neg\mathsf{EG}\ q$), and ϕ does not contain nested CTL expressions (e.g. $\mathsf{EF}(\mathsf{EG}\ p)$).

The first limitation can be easily overcome by using the three-valued equivalents of the Boolean operators (Figure 2.11): $e_{\neg\phi}(\mathbf{s}) = \neg e_\phi(\mathbf{s})$; $e_{\phi\vee\theta}(\mathbf{s}) = e_\phi(\mathbf{s}) \vee e_\theta(\mathbf{s})$; $e_{\phi\wedge\theta}(\mathbf{s}) = e_\phi(\mathbf{s}) \wedge e_\theta(\mathbf{s})$.

A heuristic was proposed to extend the three-valued evaluation to nested CTL expressions in [*a*31], we refer the reader to that document for details.

Simple termination heuristic. In this section a precise termination condition was proposed for the B-I-Sat algorithm, but its efficient implementation is a future work. For the evaluation of the B-I-Sat strategies, a simplified termination heuristic was used. Given a temporal logic formula ϕ, let us construct an equivalent ϕ' where each temporal logic operator is non-negated. This can be done using the duality rules of CTL, e.g. $\neg\mathsf{EF}\ p = \mathsf{AG}(\neg p)$. If ϕ' contains only E operators, the "true" result given based on a partial state space can be accepted as result for the complete state space, as can be seen from the discussion of the termination conditions above: $\mathbf{s} \in \mathcal{S}^b_\phi \Rightarrow e_\phi(\mathbf{s}) = T$ if ϕ is a simple CTL expression with any of the four operator pairs. The simple termination heuristic continues the exploration otherwise, therefore it gives a result "false" only if the full state space is explored. The three-valued termination condition defined above is more precise, it would allow early termination in certain cases with a false result.

Similarly, it can be shown that if ϕ' contains only A operators, the "false" evaluation result can be accepted based on a partial state space, otherwise the state space exploration continues. This simple termination heuristic does not provide a solution for a formula ϕ' that contains both A and E temporal operators.

Publications related to this section. The usage of three-valued logic and the discussed rules for termination were first drawn up in [*a*31], then refined in [*j*4].

[3] $\mathcal{F} \cap p$ denotes the set of states in $\mathcal{F}$ which satisfies p.

2.6 Evaluation

This section is dedicated to the evaluation of the proposed B-I-Sat algorithm and the three proposed strategies. After discussing the measurement considerations in Section 2.6.1, the Sections 2.6.2 and 2.6.3 provide measurements on widely-used benchmark models. Section 2.6.4 discusses the industrial use of the B-I-Sat algorithm.

2.6.1 Measurement Considerations

The goal of the work described in this chapter is to show the feasibility of an iterative bounded saturation-based CTL model checking algorithm in order to improve the performance of saturation. To assess the performance, an extensive measurement campaign was performed. The most important measurements are discussed in this chapter.

For the measurements a desktop PC (Intel Core i7-3770 3.4 GHz CPU, 8 GB memory, HGST Travelstar Z7K500 HDD with Windows 7 x64 and .NET 4.0 framework) was used, running PetriDotNet 1.5-beta3. Each run time measurement was performed 5 times. The presented results are the average of the measurements, with the minimum and maximum measured execution times excluded ($m = \frac{\sum_{i=1}^{5}(m_i) - \max m_i - \min m_i}{3}$, where m_i denotes the measured time values, $i = 1, \ldots, 5$).

Baseline. As the goal of this work was to improve the performance of the unbounded saturation, the performance comparisons will be performed with respect to the unbounded saturation-based CTL model checking, as implemented in PetriDotNet 1.5-beta3 [*c*10] (without the lazy saturation introduced in [*j*26]). The results of this algorithm will be labelled by "Unbounded".

For the benchmark models – to show the overhead of the iterative algorithms –, the non-iterative, bounded saturation (see Section 2.3.1) will also be used as baseline with the a priori knowledge of the smallest bound value (denoted by b in the measurements table) that is needed to give an answer representative to the whole state space. Obviously, the knowledge of this optimal bound should not be expected, but it shows a lower limit for the execution time of the fully EDD-based strategies (restarting and continuing). These measurements are labelled with "Fixed bound" in the measurement tables. All the bounded measurements used the exact pruning strategy with caches (as described in [*c*18]). This way the number of necessary iterations does not depend on the decomposition of the model, and it is easier to analyse the results.

The various strategies of B-I-Sat are also implemented in PetriDotNet. These implementations do not use the three-valued termination conditions. Instead a simpler method is used: the CTL expressions containing only non-negated E operators are evaluated iteratively with increasing bounds until the result is "true" or the full state space is explored (see Section 2.5.2).

Note that many of the measurements (selection of models and requirements) were inspired by earlier work (e.g. [*a*30; *j*2]), but the baselines and the implementations are different, thus the values cannot be directly compared. Besides, it is worth to note that the PetriDotNet tool is based on the Microsoft .NET framework, it is implemented in C# which is a managed language with garbage collection. This might seem to be an unusual choice for the implementation of formal verification algorithms. The details of this choice are described in [*c*10].

Performance metrics. Typically the main aspects of performance evaluation are execution time, memory consumption and I/O (disk, network) usage. The I/O usage of the presented algorithms is negligible (not including the usage of swap files, see later), therefore it will not be measured.

The memory consumption of an algorithm is primordial. However, measuring memory consumption of algorithms implemented using managed languages (using non-deterministic garbage collection) is not precise. Furthermore, usually most of the resources of the executing computer are available for the model checking algorithm. If the amount of available physical memory is not enough, the swap file will be heavily used (so-called *thrashing*). As hard disks are much slower than the memory, this is reflected in the execution time. Measurements showed that on the used configuration thrashing occurs at around 6 GB of memory consumption. The cases where timeout occurs because of thrashing will be denoted by ">6 GB" in the measurements table.

In the following the main focus is on the execution time of the various model checking algorithms. The loading time of the models is excluded from the measured execution time for all the compared algorithms.

Aspects of evaluation. In this chapter various goals, hypotheses and questions were drafted. Based on them, the focus of the evaluation should be on the following statements and questions (evaluation aspects):

EA1. The B-I-Sat algorithm *provides lower run times* for certain models than the unbounded saturation.

EA2. The B-I-Sat algorithm *scales better* for certain models than the unbounded saturation.

EA3. The B-I-Sat algorithm may be more expensive computationally than the unbounded saturation if a big part of the state space needs to be explored for the evaluation of the given formula.

EA4. The compacting strategy of the B-I-Sat algorithm can reduce the size of the state space representation in the exploration phase.

EA5. Which strategy provides the best performance for B-I-Sat?

2.6.2 Execution Time Evaluation on Benchmark Models

This section presents run time measurements on various benchmark models. These benchmark models are widely used to evaluate saturation-based techniques. The models (in a format supported by PetriDotNet) and their description can be downloaded from the PetriDotNet website[4].

Each model is scalable, and the same CTL expression was evaluated with different parameters of the models. The measurement results divided into six measurement groups are presented in Table 2.3.

- Measurements on the *Kanban* model (group (1) in Table 2.3) show clear advantage of bounded methods. For a low parameter value ($N = 30$) the unbounded algorithm was the fastest, and EDD-based methods could not beat it even if the optimal exploration depth ($b = 35$) was known a priori. As the size of the state space increases with the increasing parameter values, the B-I-Sat strategies provide much better results compared to the unbounded variant. In case of this model, the restarting strategy provided the lowest execution time.

- Measurements on the *Slotted ring* model (group (2) in Table 2.3) show similar results. In this case the requirement is very shallow, the expression may be evaluated with exploration depth $b = 8$. This is clearly a situation where bounded model checking algorithms can excel.

- Measurements on the *Hanoi* model (group (3) in Table 2.3) demonstrate a different type of model. It models the widely-known Tower of Hanoi game, where N is the number of disks. It can be

[4]The benchmark models used in this section and their documentation (including the used decompositions) are published under doi: 10.5281/zenodo.200500.

seen that the unbounded algorithm could not cope with $N \geq 14$ parameter values, as the state space exploration cannot be performed using the available memory. However, the given requirement (moving the 8th smallest disk from rod A to rod B) does not depend on the parameter N (i.e. the 9th, 10th, ..., Nth disks do not have to be moved), therefore the B-I-Sat algorithms can provide a nearly constant execution time. In this case, the restarting strategy was significantly slower, and the continuing strategy provided the best results. The compacting strategy in the previous measurement groups was about 3–4 times slower than the restarting strategy, but in group (3) it provided nearly as good execution times as the continuing strategy.

- Measurements on the *FMS* model (group (4) in Table 2.3) show a particularly interesting phenomenon. The restarting and continuing strategies provide better run time for parameter values $N \geq 1000$ compared to the unbounded algorithm. Their run time is close to the execution time of the non-iterative solution. However, the execution time of compacting saturation is more than three times less than the other B-I-Sat strategies'. This is due to the more compact, MDD-based storage.

- Measurements in groups (1)–(4) showed cases when the B-I-Sat algorithm provided shorter execution times than the unbounded algorithm. These measurements support the EA1. and EA2. statements: the B-I-Sat algorithm provides shorter run times and scales better for certain models than the unbounded saturation. Depending on the models and the expressions, different strategies provided the best results. This already shows that there is no clear answer to the question EA5., each of the three proposed B-I-Sat strategies may overcome the others in certain cases.

- Measurements in groups (5) and (6) show cases where bounded model checking provides worse performance than unbounded saturation, supporting the (nearly obvious) EA3. statement. In both cases it can be seen that even the fixed bound variant provides significantly slower execution than the unbounded algorithm. This means that it is not possible to make a purely EDD-based algorithm faster than the unbounded algorithm, independent of the values for B_0, B or how low the overhead of the iterative execution is.

  In case of the *Round Robin* model with the given requirement (group (5) in Table 2.3), the required exploration depth depends on N, making the B-I-Sat algorithms less scalable than in other cases.

  Contrarily, the given requirement on the *DPhil* model (group (6) in Table 2.3) necessitates only a shallow exploration until a constant depth of 4. However, DPhil is a highly asynchronous model with small diameter (maximum state distance). For $N = 1000$, there are only 1000 states at distance 1, but there are 8.6×10^{10} states within bound $b = 5$. To store the states within the $b = 5$ bound, 63,857 EDD nodes are needed, while storing the full state space requires only 19,977 MDD nodes. As shown later in this section, even the compacting saturation cannot efficiently reduce the state space of this model.

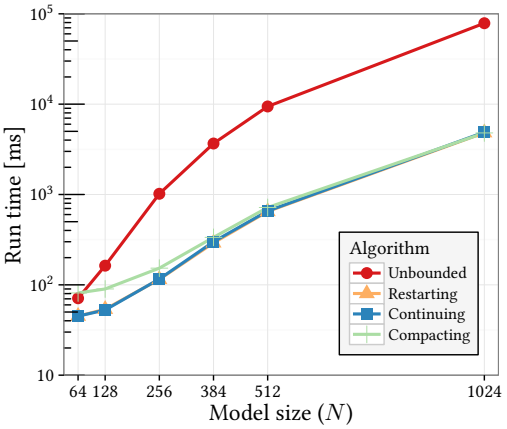
Scaling. Two measurements are presented here to discuss the scaling of the iterative methods in more detail. Figure 2.12(a) shows how the different algorithms scale with the growing model size on the *Counter–N* model (representing an N-bit binary counter). It can be seen that the unbounded algorithm requires significantly more time to execute than the B-I-Sat algorithm. The various B-I-Sat strategies provided similar run times.

Table 2.3: Run times of CTL expression evaluation

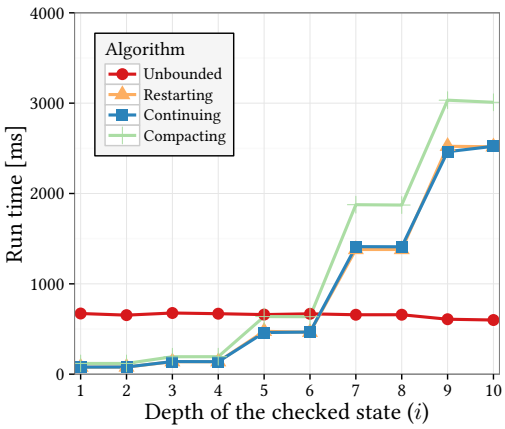
N	Run time [s] Unbounded	Restarting	Continuing	Compacting	Fixed bound
(1) Kanban–N, expression: $\mathsf{EF}(\mathit{pout}_4 = 5)$, $B_0 = B = 10$					
30	**0.34**	0.96	1.23	3.41	($b = 35$) *0.54*
50	1.74	**1.28**	1.69	3.92	($b = 35$) *0.63*
100	24.99	**1.28**	1.68	3.93	($b = 35$) *0.64*
200	>300	**1.28**	1.63	3.92	($b = 35$) *0.66*
(2) SlottedRing–N (SR–N), expression: $\mathsf{EF}(E_2 = 1 \wedge A_2 = 1)$, $B_0 = B = 5$					
100	10.64	**1.08**	1.36	3.40	($b = 8$) *0.58*
200	88.34	**3.01**	3.75	10.84	($b = 8$) *1.66*
300	>300	**5.70**	7.07	22.70	($b = 8$) *3.16*
(3) Hanoi–N, expression: $\mathsf{EG}(\mathsf{EF}(B_{N-8} = 1))$, $B_0 = B = 10$					
12	26.97	1.76	**0.59**	0.72	($b = 128$) *0.40*
14	>6 GB	2.11	**0.68**	0.79	($b = 128$) *0.48*
16	>6 GB	2.40	**0.80**	0.89	($b = 128$) *0.50*
18	>6 GB	2.70	**0.95**	**0.98**	($b = 128$) *0.57*
20	>6 GB	3.15	**1.04**	**1.06**	($b = 128$) *0.78*
(4) FMS–N, expression: $\mathsf{EG}(\mathsf{E}(\mathit{M1} > 0\ \mathsf{U}\ (\mathit{P1s} = \mathit{P2s} = \mathit{P3s} = 3)))$, $B_0 = B = 10$					
25	**0.92**	29.00	30.39	4.77	($b = 30$) *28.13*
50	5.44	44.58	46.19	**5.28**	($b = 30$) *43.42*
100	45.75	46.74	49.11	**5.25**	($b = 30$) *44.93*
1000	>6 GB	46.88	50.39	**5.28**	($b = 30$) *44.72*
10,000	>6 GB	47.82	49.79	**5.28**	($b = 30$) *46.07*
(5) Round Robin–N (RR–N), expression: $\mathsf{EG}(\mathit{true})$, $B_0 = B = 10$					
10	**0.09**	0.12	0.11	0.50	($b = 39$) *0.04*
25	**0.38**	2.61	1.59	17.53	($b = 99$) *1.00*
50	**2.05**	51.91	21.87	>300	($b = 199$) *9.67*
(6) DPhil–N, expression: $\mathsf{E}(\neg \mathit{eating}_2\ \mathsf{U}\ \mathit{eating}_1)$, $B_0 = B = 10$					
10	**0.08**	**0.09**	**0.09**	0.26	($b = 4$) *0.02*
50	**0.14**	0.62	0.64	2.12	($b = 4$) *0.11*
100	**0.24**	1.52	1.61	4.66	($b = 4$) *0.31*
1000	**2.21**	78.98	78.71	>300	($b = 4$) *18.21*

The requirement in this case checks the reachability of a state where the 12th bit is 1, i.e. it checks whether it is possible to count up to 2^{12}. One could expect that in this case the execution time of the B-I-Sat algorithm should be constant. While this seems to be intuitive (i.e. bits $n > 12$ can stay constantly 0), unfortunately this is not necessarily true. As the number of places grows, the potential state space to be encoded grows too, making the decision diagrams bigger. This reduces the performance of the saturation algorithms. By using the knowledge about the model and the requirement, more sophisticated decomposition (e.g. keeping all bits $n > 13$ in one component) can be done, resulting in a nearly constant execution time, around 0.4 s for each value N using the restarting strategy. It has to be noted that the Petri net models of Counter–N are huge: the model with $N = 1024$ contains 1024 places, 1025 transitions and 526,848 edges. The compared implementations are not optimised for such high amount of edges, causing a long model loading time.

A second scaling measurement in Figure 2.12(b) targets a case when the model remains the same, but a parameter in the requirement changes, causing different required exploration depths. The model used as an example here is the *Queen–10* model, where 10 queens should be arranged on a 10×10



(a) Execution time of evaluating $\mathsf{EF}(bit_{12} = 1)$ on Counter–N models ($B_0 = B = 16$)



(b) Execution time of evaluating $\mathsf{EF}(q_i = 0)$ on Queen–10 model ($B_0 = B = 2$)

Figure 2.12: Scaling measurements

chessboard without any two threatening each other. It is obvious that there should be exactly one queen per row (rank). In the model first a queen is placed in the first row, after the next to the second row, etc. A place q_i has a token only if no queen is placed yet to the row $i \in \{1, \ldots, N\}$. Therefore checking the reachability of $q_1 = 0$ is trivial (equivalent to placing one queen to the first row of an empty chessboard), while the reachability of $q_N = 0$ is equivalent to checking if the N-queens problem can be solved or not.

As can be seen in Figure 2.12(b), the B-I-Sat algorithm provides lower execution times for the shallow requirements, up to $\mathsf{EF}(q_6 = 0)$. After, the iterative bounded methods are becoming slower than the unbounded saturation, which provides nearly constant run time, independently of the parameter i in the requirement. In these measurements, $B_0 = B = 2$ bound values were used. The effect of the $B = 2$ can be seen on the figure as plateaus. For example, checking $\mathsf{EF}(q_5 = 0)$ and $\mathsf{EF}(q_6 = 0)$ takes nearly the same amount of time, as the execution time is dominated by the state space exploration in this case and the same number of iterations are required for the evaluation of both expressions. This illustrates that bounded algorithms have an advantage compared to other methods typically when the problem is shallow, i.e. the given requirement can be evaluated by exploring the model only up to a small depth.

2.6.3 Memory Consumption Evaluation on Benchmark Models

As it was discussed at the beginning of the section, the measurement of the memory consumption is not in the main focus of the current evaluation. Interested readers may find more detailed memory consumption measurements in [*a*30]. Here some of the measurements from [*a*30] are repeated.

Measuring the peak or average memory consumption of managed language implementations may be imprecise and misleading. Here instead we focus on measuring the number of EDD and MDD nodes created that is representative for the memory requirements of the different algorithms. It has to be noted that the "logical" creation of a new node does not necessarily lead to a constructor call: a pool of node objects is kept in the current implementation. When a node object is no longer needed, it will be put into the pool. If possible, instead of creating a new node, an old node from the pool is reused. Earlier measurements showed that a pool with moderate maximum capacity (500 in this case)

significantly reduces the number of object instantiations (e.g. by a factor of 6 in case of the FMS–5 model) and improves the performance of the implementation [*a*30].

In the following, the three proposed strategies of B-I-Sat will be compared to the MDD-based unbounded algorithm ("Unbounded" column) and to the EDD-based non-iterative model checking with infinite initial bound ("Bounded ($B_0 = \infty$)" column). Both of these algorithms will explore the whole state space independently from the requirement.

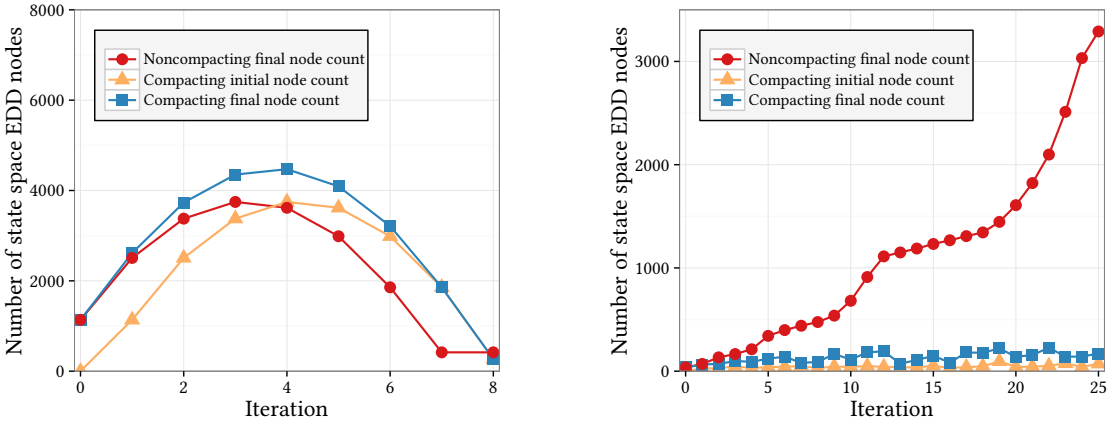
For each EDD-based algorithm two metrics are given: the number of created EDD and MDD nodes. EDD nodes are created during the bounded state space exploration. MDD nodes are created when the EDD is converted to MDD for CTL evaluation, also during the CTL evaluation itself. The MDD node creation counts exclude the MDD nodes created to represent the next-state function. For the unbounded algorithm only the number of created MDD nodes is given as it creates no EDD nodes.

The measurements for three different model groups are presented in Table 2.4. It has to be noted that these values show the number of constructor calls and they were obtained with pooling (with capacity of 500) that lowers the number of needed constructor calls. The following observations can be made based on the table of measurements.

- The bounded non-iterative algorithm creates more EDD and MDD nodes in total than the unbounded algorithm. This shows that in general the EDD-based state space representation is less compact.
- As the restarting and continuing strategies lead to the same EDD representation in each iteration, the numbers of MDD nodes created by these two strategies are identical.
- The compacting strategy creates more MDD nodes than the other B-I-Sat strategies. However, in the measurement groups (1) and (2) in Table 2.4, the compacting strategy leads to significantly less EDD nodes created than the restarting or continuing strategy.
- In case of the *Hanoi* model with the given requirement (group (1) in Table 2.4), the continuing strategy results in much less EDD nodes created, because it was able to efficiently reuse the EDD nodes from previous iterations. However, in the measurement groups (2) and (3) it could not benefit from the previous iterations, leading to slightly more EDD nodes created than by the restarting method. It can also be observed that the B-I-Sat algorithms created similar number of nodes independently of the parameter N, while the non-iterative algorithms could not cope with the increasing N values.
- In case of the *Queen–10* model with the given requirements (group (3) in Table 2.4), the node counts of the unbounded and non-iterative bounded algorithms are constant or slowly increasing, while the iterative B-I-Sat algorithms need significantly more nodes as the depth of the requirement increases. This is in accordance with the run time measurements presented in Figure 2.12(b).

These measurements also support the previous findings that it varies which strategy provides the best result depending on the model and the requirement (EA5.). It is also visible that where the compacting saturation provided good performance (e.g. in case of Hanoi and FMS models according to Table 2.3), the number of created EDD nodes is lower by at least a factor of 10 than the EDD nodes created by the other strategies. In other cases, the compacting saturation cannot reduce significantly the size of EDD representation.

The observed difference is caused by the different characteristics of the state spaces. Figure 2.13 shows the evolution of the EDD representation size over the iterations for two different models. In case of the *DPhil* model, the state space is relatively wide (there are many states with low distance values) and shallow (there are few states with high distance values). In Figure 2.13(a) it can be seen that the EDD representations of the different strategies lead to similar sizes. In case of the *Hanoi* model, the characteristics of the state space are different (see Figure 2.13(b)). As the state space is relatively



(a) Size of state space EDD of DPhil–20 model

(b) Size of state space EDD of Hanoi–8 model

Figure 2.13: Size of state space EDD

narrow and deep, the compacting strategy succeeds to reduce the EDD node count in each iteration, keeping it continuously low. The other B-I-Sat strategies cannot do this, which results in a steadily growing state space representation. This clearly shows that the compacting strategy can reduce the size of the state space representation in the exploration phase in certain cases (EA4.).

Table 2.4: Total number of created node objects

N	**Total number of created node objects**								
	Unbounded	**Bounded ($B_0 = \infty$)**		**Restarting**		**Continuing**		**Compacting**	
	MDD	EDD	MDD	EDD	MDD	EDD	MDD	EDD	MDD
(1) Hanoi–N, expression: $\mathsf{EG}\,\mathsf{EF}(B_{N-8} > 0)$, $B_0 = B = 10$									
8	124,046	108,767	55,391	124,279	16,094	37,874	16,094	3,493	29,198
10	1,083,002	2,126,550	373,434	122,500	16,432	37,992	16,432	3,571	27,833
12	7,863,690	—	—	122,554	16,613	38,022	16,613	3,623	27,754
16	—	—	—	122,662	16,941	38,082	16,941	3,727	27,986
(2) FMS–N, expression: $\mathsf{E}[M1 > 0\ \mathsf{U}\ P1s = 3 \wedge P2s = 3 \wedge P3s = 3]$, $B_0 = B = 10$									
25	50,647	17,790	91,538	770,299	87,699	774,300	87,699	68,491	92,361
50	198,255	110,506	346,688	792,204	88,244	796,302	88,244	68,972	92,993
100	777,530	783,221	1,350,113	792,224	88,244	796,322	88,244	68,972	92,993
(3) Queen–10, expression: $\mathsf{EF}(q_N = 0)$, $B_0 = B = 2$									
2	181,217	87,455	161,362	3,197	626	3,200	626	3,150	626
4	181,529	87,455	161,361	12,052	4,260	10,802	4,260	7,392	4,001
6	184,703	87,455	161,360	48,931	37,201	45,276	37,201	34,620	35,124
8	206,689	87,455	161,359	127,304	180,227	120,368	180,227	84,564	163,913
10	248,754	87,455	161,358	214,201	436,467	203,434	436,467	120,128	366,568

2.6.4 Industrial Case Study[5]

To evaluate the usability of the B-I-Sat algorithm and the different strategies on industrial examples, measurements were performed on a model describing a real, industrial safety function. The case study is the verification of the PRISE (primary-to-secondary leakage) safety logic, a safety function included in the Reactor Protection System of a nuclear power plant [NB09]. This safety function initiates an emergency operation if a predefined chain of events happens. The detection of the specific event chain requires a complex logic, the design of which is error-prone. This also puts emphasis on the necessity of using formal verification to ensure correctness.

The safety function receives inputs from different sensors, and computes the values of outputs, one of which initiates the emergency protection action. The values of the outputs depend on the recent and past values of the inputs, and some internal timers. The design of the controller was specified by simple combinatorial (OR gates, AND gates, and inverters), and sequential (SR flip-flops, delay and pulse modules) function blocks, similar to the blocks defined for the FBD language in [I61131-3]. The proper combination of these logic elements is required to guarantee that the emergency protection action will be initiated only in case of a specific, dangerous event happened.

A coloured Petri net (CPN) model of this safety logic was created in [*c*28]. The structure of the CPN model preserves the data flow characteristics of the function block description. The model can be parametrised – the parameters are the delay or pulse durations of timers. The first successful verification attempt of this safety function was presented in [*c*28], using saturation-based CTL model checking. Previous attempts to use model checking on the complete model of the safety logic have failed[6] [Tót09].

Here measurements are presented for four different parametrisation of the models (denoted by PRISE S, M, L, and XL). The safety logic follows a cyclic execution schema. In the proposed model each execution cycle consists of 29 transition firings. Earlier work [*a*30] showed that this is one of the best choices of $B_0 = B$ for this model.

The measurements being presented in Table 2.5 show that if the verification requirement is shallow (measurement groups (1), (2) and (3)), then the bounded algorithms provide significantly lower execution time than the unbounded algorithm. The different requirements are shallow for different reasons. The requirement in measurement (1) targets an output that does not depend on the complex behaviour of the block. For the measurement (2) a fault was injected into the design: an OR gate was replaced by an AND gate, making the given requirement unsatisfied. Both of these requirements can be evaluated in the first iteration, with bound $B_0 = 29$.

In measurement (3) the requirement was not satisfied, which can be determined by exploring only the first couple of cycles of operation. However, in this case the requirement was incorrect, thus it does not have to be satisfied. This measurement group demonstrated that formal verification can help to improve the understanding of the analysed systems. However, it also pointed to a weak point of this approach: understanding the cause of the model checker's result and fixing the requirement took about 3–4 man-hours of two engineers experienced both in formal verification methods and the model of the PRISE system. Improvements to the requirement handling and methods to improve the presentation of the results will be presented in the following chapters.

Another interesting phenomenon can be observed in the measurement group (3): as the model parameters grow, the run time of the B-I-Sat algorithms decreases. This is because the models with bigger parameter values have longer delay timer values and for the models with larger parameters

[5]The introduction and the discussion of the model in this subsection is an adaptation of Section 5.2 of [*j*2]. The presented measurements are new.

[6]The authors of [NB09] decomposed the system and have done manual compositional verification.

the requirement can be evaluated before the timer expires. Therefore longer delay times imply less explored behaviours and smaller bounded state space to be checked in these cases.

The measurement group (4) targets a complex behaviour. As it is a safety requirement (invariant property), it is satisfied and it concerns the full model, the B-I-Sat algorithms can only give answer based on the exploration of the full state space. In this case it is not possible to benefit from the advantages of bounded model checking and the overhead of the less compact state space storage and the overhead of iterations make these B-I-Sat algorithms not competitive with the unbounded saturation. However, it is worth to notice that the compacting strategy provided a better result than the other strategies.

These measurements show that bounded model checking may reduce the resource needs of model checking in the early phases of design, when incorrect behaviour and imprecise requirements can be expected. Later, when confidence is gained in the design and finding many faults is not expected, unbounded model checking may provide a better overall result.

The PRISE safety logic was implemented using the TELEPERM XS platform of Areva [NB09], which targets specifically the control tasks of nuclear power plants. As the goal was to verify a single safety logic (the rest of the control system is stable), it was a more efficient choice to use direct modelling instead of developing a dedicated workflow for the verification of programs written for this platform, although obviously this choice implies the needs for experts in modelling and verifying the safety logic. Furthermore, using the direct modelling it was possible to omit the detailed analysis of certain aspects of the semantics of the TELEPERM XS platform. Instead, all possible behaviours were modelled (e.g. all possible execution order of the concurrent FBD blocks), thus if the safety requirements are satisfied on this model, the result will hold for the real implementation with the real semantics too.

B-I-Sat for test input generation. Besides the verification of the PRISE safety logic, the B-I-Sat algorithm was reused for test input generation for laser-guided vehicles in the R3-COP Artemis project [*e*20]. In this project we have analysed a coloured Petri net modelling the communication protocol between the central traffic management computer and the vehicles. The B-I-Sat algorithm was used to generate test input sequences for robustness testing. The input of the test sequence generation is a set of CTL requirements $R = \{r_1, r_2, \ldots, r_n\}$. The goal of the algorithm is to find a state sequence $U = (\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m)$, where $\exists i_1, \ldots, i_n \colon \mathbf{s}_{i_1} \vDash r_1, \ldots, \mathbf{s}_{i_n} \vDash r_n$ and $i_1 \leq \cdots \leq i_n$, i.e. a trace that goes though certain states, each of them satisfying one (or some) of the given requirements. A relaxed version of this problem is when the requirements are not ordered, thus the generated path should satisfy only $\mathbf{s}_{i_1} \vDash r_1, \ldots, \mathbf{s}_{i_n} \vDash r_n$, but there is no restriction on the order of $i_1, \ldots, i_n$.

To generate the test sequences, the B-I-Sat algorithm was used iteratively, in a greedy manner. In the ordered case, first the algorithm was looking for the state $\mathbf{s}_{i_1}$ satisfying requirement r_1 that is closest to the initial state $\mathbf{s}_0$. Next, based on the EDD encoding of the state space and the $\mathcal{N}^{-1}$ relation a trace was extracted between $\mathbf{s}_0$ and $\mathbf{s}_{i_1}$. After, the algorithm is restarted from $\mathbf{s}_{i_1}$, looking for the closest $\mathbf{s}_{i_2}$ satisfying requirement r_2, etc. The algorithm does not provide a good approximation of the optimum, but in practice it was found to be useful [*e*20].

Publications related to this section. The implementation of the B-I-Sat algorithms in the PetriDotNet framework was discussed in [*c*10]. Similar measurements with different focus were presented for the B-I-Sat algorithms in [*j*4; *j*2; *a*30]. The verification of the PRISE safety logic with saturation-based techniques was first reported in [*j*26]. The application of B-I-Sat algorithm to test generation was discussed in [*e*20] which also provides the pseudocode of the various trace generation strategies.

Table 2.5: Run times of CTL expression evaluation on PRISE models

Model	Run time [s]			
	Unbounded	**Restarting**	**Continuing**	**Compacting**
(1) *The ACTIVE output (OUTPUT-2) can be true.* ($B_0 = B = 29$)				
PRISE S	0.62	**0.36**	**0.37**	0.45
PRISE M	1.38	**0.36**	**0.37**	0.45
PRISE L	2.35	**0.37**	**0.36**	0.45
PRISE XL	155.40	**0.45**	**0.46**	0.53
(2) *If the stream generator is in an inhibited state (INPUT-8=true), then the RS-FF block connected to OUTPUT-1 should be reset. (Fault injected into the models: OR gate connecting INPUT-8 and INPUT-9 was replaced by an AND gate.)* ($B_0 = B = 29$)				
PRISE S (faulty)	0.98	**0.36**	**0.37**	0.49
PRISE M (faulty)	2.20	**0.36**	**0.37**	0.49
PRISE L (faulty)	3.73	**0.37**	**0.37**	0.50
PRISE XL (faulty)	166.68	**0.44**	**0.45**	0.58
(3) *If INPUT-5 is false and the connected pulse timer is not on, the pulse timer's output will be true in the next cycle. (Incorrect requirement, not satisfied.)* ($B_0 = B = 29$)				
PRISE S	**1.81**	2.26	2.39	4.64
PRISE M	3.90	**1.51**	1.58	2.95
PRISE L	6.45	**1.53**	1.57	2.95
PRISE XL	188.94	**1.67**	1.75	3.09
(4) *If there is an emergency action (OUTPUT-1=true), then the steam generator water was high (INPUT-1) for* $\geq t_1$ *time and the primary pressure was decreasing (INPUT-2) for* $\leq t_2$ *time.* ($B_0 = B = 29$)				
PRISE S	**0.84**	33.20	19.29	11.52
PRISE M	**1.90**	430.10	114.36	30.05
PRISE L	**3.31**	>6 GB	424.14	52.41
PRISE XL	**164.81**	>6 GB	>6 GB	>600

2.7 Summary and Future Work

This chapter introduced and evaluated B-I-Sat, a bounded model checking algorithm based on the saturation-based techniques. It demonstrated the feasibility of building an efficient model checking algorithm on saturation, reusing the ideas of bounded model checking. Three different strategies were proposed for B-I-Sat: the restarting, continuing and compacting strategies. Their main properties are summarised in Table 2.6.

Table 2.6: Comparison of the three proposed strategies for B-I-Sat

	Restarting	**Continuing**	**Compacting**
States explored in iteration i	$\mathcal{S}_{[0;i\cdot B]}$	$\mathcal{S}_{[0;i\cdot B]}$	$\mathcal{S}_{[(i-1)\cdot B;i\cdot B]}$
Initial state set in iteration $i > 1$	$\mathcal{S}_0$	$\mathcal{S}_{[0;(i-1)\cdot B]}$	$\mathcal{S}_{[(i-1)\cdot B;(i-1)\cdot B]}$
State set for model checking in iteration i	$MDD(\mathcal{S}_{[0;i\cdot B]})$	$MDD(\mathcal{S}_{[0;i\cdot B]})$	$\bigcup_{j=1}^{i} MDD(\mathcal{S}_{[(j-1)\cdot B;j\cdot B]})$

The discussion and the evaluation showed that B-I-Sat may reduce the resource needs of model checking compared to the unbounded saturation-based model checking algorithm in certain cases, leading to shorter execution times and a greater set of verifiable models. Various examples were shown where B-I-Sat scales better than the unbounded algorithm. It was highly dependent on the model and the requirement which B-I-Sat strategy provided the best performance, therefore there is no clear

best strategy among the three proposed strategies. The bounded algorithms may be more resource consuming if a big part of the model has to be explored for the evaluation of the requirements.

The contributions targeted in this chapter were the following.

Thesis 1 I designed *B-I-Sat* (Bounded Iterative Saturation), a novel computation tree logic (CTL) model checking algorithm, that efficiently combines bounded model checking with saturation-based techniques.

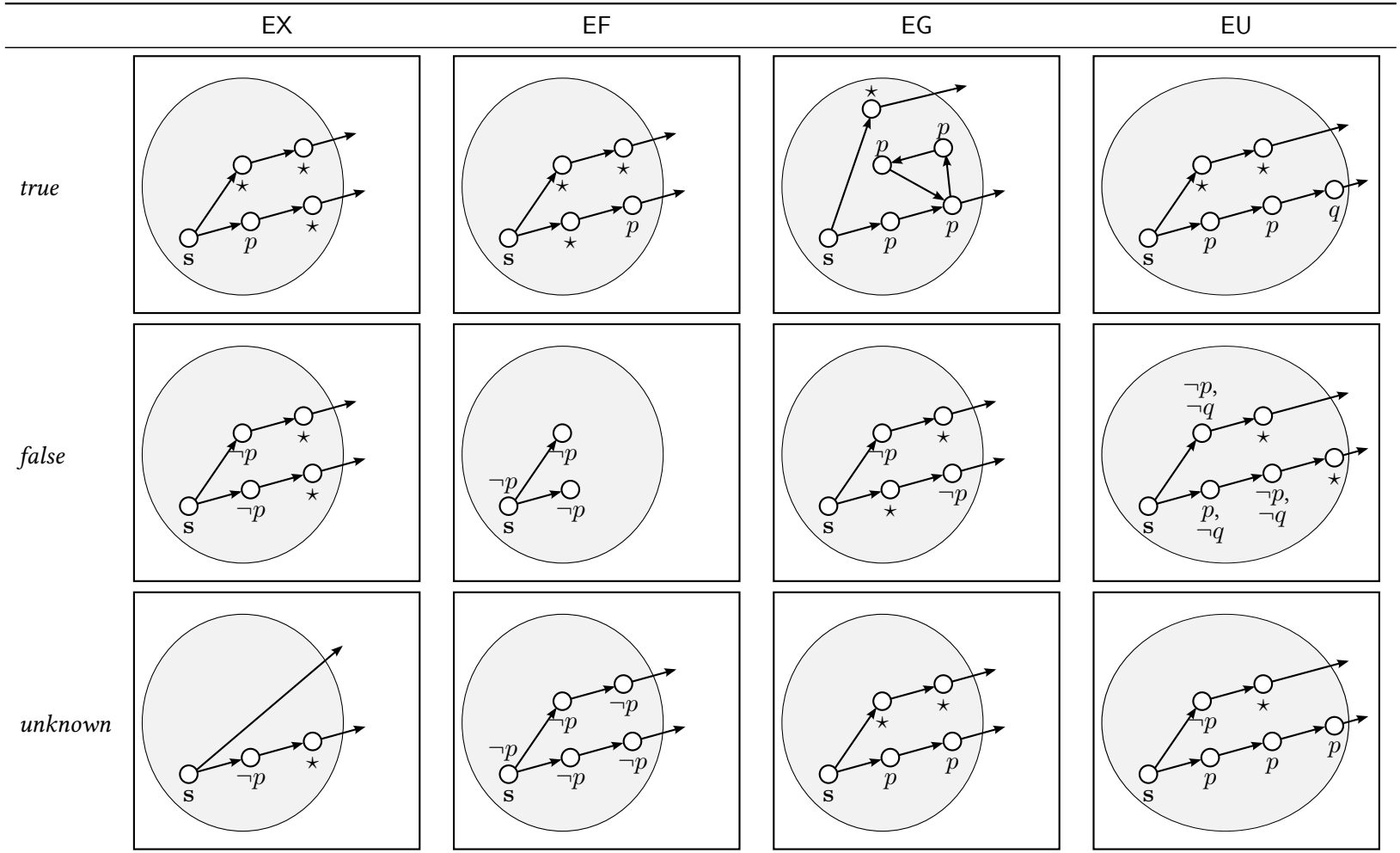
- 1.1 I defined the building blocks, and based on them the B-I-Sat algorithm to perform bounded CTL model checking using saturation-based techniques. I defined two strategies for B-I-Sat: the restarting and continuing strategies.
- 1.2 I defined termination conditions for the B-I-Sat algorithm using three-valued logic.
- 1.3 I developed an advanced incremental search strategy, the so-called compacting strategy to reduce the memory consumption of the B-I-Sat algorithm.
- 1.4 I evaluated the performance of the B-I-Sat algorithm with the different strategies on various benchmark models and an industrial example.

Thesis 1.1 was discussed in Sections 2.2.2 and 2.3. The termination conditions based on three-valued logic (Thesis 1.2) were introduced in Section 2.5. Section 2.4 proposed the compacting strategy (Thesis 1.3). The evaluation of the different B-I-Sat algorithms (Thesis 1.4) were described in Section 2.6.

Future work. There are four main research directions for the future work, as follows.

- As it is highly model and requirement-dependent if B-I-Sat provides better execution time than the unbounded saturation algorithm and which B-I-Sat strategy is the most efficient, future work is required on heuristics that may suggest strategies for the user.
- The measurements presented here used simple, model-based, predefined decomposition strategies. The decomposition may highly affect the performance of the saturation-based techniques, automated decomposition heuristics would be needed.
- A future research direction targets the generalised implementation of the presented algorithms to make them applicable to non-Petri net models.
- Proposing an efficient implementation for the three-valued termination condition is also a future work.

Table 2.7: Examples of the three-valued evaluations based on the partial state space (based on [*a*31])



Chapter 3

Model Checking Critical PLC Programs

The advantages of model checking were already advocated in the previous chapter. To introduce model checking to the development of industrial control software and widely use them (not only in highly critical cases), manual formalisation of requirements and implementations should not be required. To hide the formal requirements and formal models, automated methods are needed to generate these artefacts based on inputs that can be easily provided and understood by the targeted users. Furthermore, as the formal model should be hidden, the often needed manual optimisations should be automated as well.

In the frame of this work a model checking-based verification workflow was designed, specifically targeting the verification of programmable logic controller software. Although model checking has been used since the 1980s [Cla08], generally its usage still needs significant effort, special knowledge and expertise, and a lot of manual work. After targeting the performance aspects of model checking in Chapter 2, this work focuses mainly on the challenges of real-life usability.

Goal. The high-level goal of the work described in this chapter is to propose a model checking solution for PLC programs that can be used in practice by PLC developers without excessive effort or assistance from formal verification experts. This was a joint work mainly with B. Fernández [*j*3]. This chapter discusses four main subgoals that were my own contributions: *(i)* proposing intermediate representations for the verification workflow, *(ii)* designing reduction heuristics to reduce the resource needs of the workflow, *(iii)* adapting the verification workflow to be applicable to safety PLCs, and *(iv)* implementing the complete workflow.

Structure of this chapter. The structure of this chapter is as follows. Section 3.1 introduces the preliminaries and the background of the work presented here. Section 3.2 provides an overview of the requirements and challenges related to the verification workflow. Section 3.3 is dedicated to the design of intermediate representations for the verification workflow. After that, Section 3.4 describes the verification workflow that was designed based on the proposed intermediate model representations. Next, the verification model reduction techniques are described in Section 3.5. This is followed by the discussion of extensions to support the safety-critical PLC programs, in Section 3.6. PLCverif, the tool implementing the discussed verification workflow, is briefly introduced in Section 3.7. In Section 3.8 usage examples are presented to demonstrate the applicability of the proposed methods in real-life development. The related work overview is provided in Section 3.9. The chapter is concluded by Section 3.10 that provides a summary and discusses future work.

3.1 Preliminaries

A *control system* is "a combination of devices and components connected together [...] to command, direct or regulate itself or another system" [Bha13]. We call *process* (or *plant* in some cases) the part of the system that is to be controlled [Bha13].

In case of non-trivial industrial plants the direct manipulation of the process is not feasible, or at least not safe and/or not economical [Par03]. *Industrial control systems* (ICS) are widely used to monitor and control industrial processes based on a pre-defined control logic, the information from the sensors and the commands of the operators. A simplistic view of this is in Figure 3.1. The operator typically receives information and gives commands through a man-machine interface, often part of a supervisory control and data acquisition (SCADA) system which is connected to the controller. This chapter targets the controller part of the industrial control systems. The controller can be implemented in different ways, e.g. using relay-based systems, but here we focus on implementations based on programmable control devices, so-called programmable logic controllers, more specifically on the software defining their application-specific behaviour.

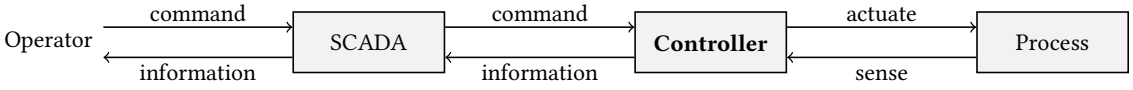


Figure 3.1: Simplified view of a control system (partially based on [Par03])

3.1.1 Programmable Logic Controllers

Programmable logic controllers (PLCs) are robust industrial computers, optimised for control tasks and the industrial environment [Bol15]. They have gained more and more usage since their introduction in 1969 [Bol15; Par03]. PLCs are now widely used for industrial control and automation tasks.

Execution schema. Most PLCs have a cyclic execution schema. In each so-called *scan cycle* the PLC *(i)* reads the input values from the physical inputs to the memory (which are then kept stable), *(ii)* executes the user program reading and modifying the memory contents, and *(iii)* writes the computed values to the physical outputs (which are then kept stable). This means that the user program observes a consistent, stable image of the inputs and the intermediate (transient) values cannot be observed on the physical outputs of the device. In addition, PLCs may have interrupts and their handling can interrupt the cyclic execution of the user code, however in this work interrupts are not targeted.

Programming languages. The way to program PLCs is standardised in the IEC 61131 standard [I61131-3], first issued in 1992 (under the name IEC 1131) that unified the pre-existing programming methods. It defines five languages: Structured Text (ST), Instruction List (IL), Function Block Diagram (FBD), Ladder Diagram (LD), and Sequential Function Chart (SFC).

As Siemens is a market leader in the field of automation[1], also Siemens is the main vendor of the systems providing the motivation of this work, we are focusing on their variants of the programming languages. Siemens PLCs support five languages that are similar to the IEC 61131 standard languages: Structured Control Language (SCL; corresponding to ST), Statement List (STL; corresponding to IL), Function Block Diagram (FBD), Ladder Diagram (LAD), and S7-GRAPH/Sequential Function Chart

[1] According to Siemens' own websites, e.g. https://www.industry.siemens.com/verticals/global/en/chemical-industries/pages/process-automation.aspx, accessed on 26/07/2016.

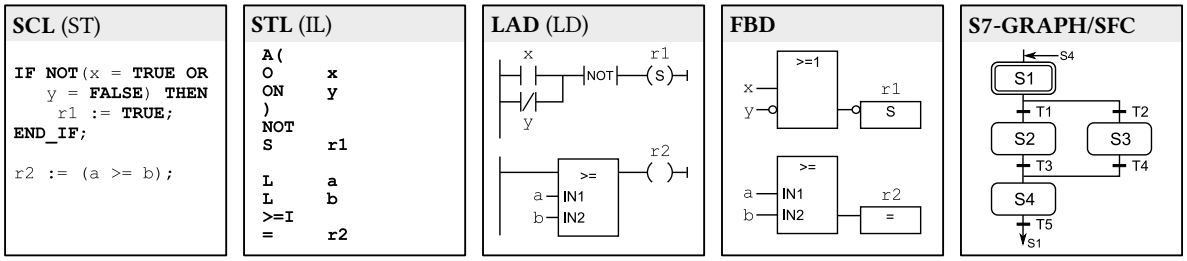


Figure 3.2: Examples of PLC languages [*c*11]

(corresponding to SFC) [Sie11]. The LAD and FBD languages are nearly identical to their corresponding standard versions, SCL contains some significant extensions (e.g. it supports jump statements, unlike the standard ST language), while STL follows rather different philosophies compared to IL (e.g. IL uses a single accumulator, STL assumes various accumulators, status words, internal stacks etc.). Figure 3.2 presents small program snippets showing some of the main characteristics of the five languages. The first four examples are execution equivalent: for any input value combination they will provide the same output values, i.e. they describe the same behaviour. The last example, written in SFC is different, as this language is more specialised than the others.

In the following, languages will be referred by their abbreviations used for the Siemens implementations (SCL, STL, LAD, FBD, SFC).

3.1.2 Motivation

CERN, the European Organization for Nuclear Research[2] operates the world's largest particle physics laboratory. A particle accelerator complex, including the 27-km-long *Large Hadron Collider* (LHC) is used to produce high-energy particle beams for dozens of experiments and facilities. Many machines depend on various industrial control systems, which are controlling e.g. the vacuum, cryogenics, or gas mixture systems. These control systems are critical for the operation: a failure of a subsystem can cause outage for the whole accelerator complex. Many of these control systems are based on PLCs, making the quality assurance of PLC programs a high priority. While the motivation of this work originates from CERN, we believe that the challenges experienced are more generic and the proposed solutions are applicable in general to the PLC-based control systems.

The most widely used verification method applied to PLC-based systems is testing, mainly acceptance testing with or without hardware in the loop, as for example for module (unit) testing there is typically no specific built-in support in the development environments [Dub11]. While testing may provide a cost-efficient way for quality assessment and improvements, exhaustive testing in general is impossible due to the excessive number of possible combinations. Additional verification methods, such as code inspection, static analysis, abstract interpretation, theorem proving or model checking [DKW08] may improve the verification process.

Formal methods were successfully applied in e.g. avionics [Wie+12], railway [Cim+12], space [Hax10] or nuclear [BS93] industries. The common in all these usages is that a failure is potentially catastrophic, it may be a threat to multiple human lives. Intel also reported about the usage of formal verification for their processors [Kai+09]. While a failure of such a CPU is likely not to cause any

[2] http://home.cern/

accident, but with hundreds of millions of CPUs sold each year [Int14] a recall campaign would cause excessive economic loss to the manufacturer.

Many of the industrial control systems are *basic process control systems* (BPCS) [I61511-1], where safety (personal and machine protection) is ensured by separated, dedicated systems (e.g. hard-wired systems, physical barriers), thus their criticality and the cost of the failure is lower. An additional important aspect is that most control systems are uniquely designed, heavily adapted to the specific process, making the "per unit" verification effort high.

These facts imply that such high verification cost as in the domains above cannot be justified for many of the industrial control systems. The usage of formal methods is typically only mandatory for highly safety-critical systems (SIL4) [SS11], but this does not mean that formal verification could not improve the quality of less critical systems. To apply formal verification to PLC-based industrial control systems (and especially to BPCS), easy-to-use, practice-oriented methods are needed that can be used with much less effort; methods that are affordable.

Model checking was already introduced in the previous chapter. This is a technique that seems to be suitable to improve the current verification practices for several reasons.

- It is an *automated method*, thus it can be the foundation of a push-button verification method.
- It can provide a *diagnostic trace* (counterexample or witness) that gives information to the users about the problem found.
- It can find *deep, hidden flaws* in the design or in the implementation which are difficult to locate by testing. Finding and correcting these problems increases the dependability of the systems.

Model checking was already applied to PLC programs, the related work is reviewed in Section 3.9.

3.2 Design of the Verification Workflow

This section overviews the requirements and the main characteristics of the designed solution for automated model checking of PLC programs [*j*3]. The details of the proposed model checking workflow will be discussed in Section 3.4. This section overviews the motivation and the background of this work. The work described in this section is a joint work of B. Fernández and D. Darvas.

3.2.1 Challenges

The main challenges related to an automated, user-friendly model checking method for PLC programs is summarised below (based on [*c*8]).

- It is difficult to use various model checkers without extensive knowledge.
- Even if the models are generated from the source code automatically, they are often too large to perform their formal verification.
- Most often no formal requirements are available, and it is difficult to use temporal logic to describe the requirements that the developer wishes to check.

3.2.2 Designing the Workflow

Requirements towards the approach. To make model checking efficiently applicable in the PLC development process, an automated, domain-specific procedure is needed that can be used *by the developers*, *without excessive training*. It has to be *supported by a dedicated tool*. Additionally, the development effort should be kept reasonably low. The original requirements were targeting *SCL programs* only, as this is the dominant PLC programming language at CERN. The verification workflow was later extended to other languages, see Section 3.6.

Input and output artefacts. It is important to choose first the input and output artefacts of the verification workflow, as this will highly constrain the solution. Model checking in general requires a model of the system and a formal requirement, and provides a result with a witness or a counterexample (if one exists).

There are no available formal models of the PLC programs we are aiming to verify. Most manually written PLC programs are developed based on informal specifications. Therefore it seems that the only possibility is to build the workflow on the PLC program. As a PLC program is considered as a precise description of the behaviour of the implementation, it is feasible to translate the implementation into a formal model.

Many model checkers require CTL and/or LTL expressions describing the requirement to be verified. Using CTL or LTL in our workflow would violate the high-level design requirement stating that the workflow should be usable for the developers without excessive training. Therefore the workflow should contain built-in support to formalise the requirements (see the details below).

The result (satisfied or not satisfied) and the diagnostic trace (witness or counterexample) can provide useful information to the users about the requirement. However, the diagnostic traces provided by the model checkers are often too long or too detailed, also they depend on the modelling of the checked implementation. Therefore the raw outputs of the model checkers are not suitable for the non-expert users, instead a self-contained, domain-specific, reduced verification report should be provided as the output of the verification workflow.

Formalising the requirements. The model checkers cannot work with informal requirements, but the users should not be exposed to temporal logic expressions without extensive training. There are various possibilities to hide the temporal logic requirements from the user. Some authors used restricted natural languages to express temporal logic formulae [Din+06; HK99]. Others use requirement patterns, e.g. [DAC99; CMS08]. We have chosen to use the latter, as requirement patterns seem to be more usable by inexperienced users. The approach we have followed in our verification workflow is similar to the one proposed by Campos *et al.* [CMS08], but more complex patterns have been created, e.g. supporting state changes, to cover the real-life needs [c16]. As examples, a simple and a complex safety requirement pattern are shown in Figure 3.3. In the examples $\mathcal{A}, \mathcal{B}, \mathcal{C}$ denote placeholders of Boolean expressions; EoC is a proposition true only at the end of the PLC's scan cycle.

Using external model checkers. To limit the development resources required for the implementation of the verification workflow (as discussed previously), it was decided at the beginning that no new model checkers will be developed. Instead, the verification workflow should be based on existing, general-purpose model checkers, such as NuSMV/nuXmv, UPPAAL[3] or ITS Tools.

The model checkers have different expressivity, different input-output formalisms, different advantages and disadvantages, different strengths and weaknesses. Furthermore, the performance of the model checkers may highly depend on the given models and requirements. Therefore it is not possible to select a single model checker that will provide the best performance and best results in every case. Instead, the verification workflow should be designed to support multiple external model checkers, which can be selected based on the current needs.

Supporting multiple model checkers may significantly increase the development needs of the workflow. This issue will be addressed in Section 3.3.

[3]It has to be noted that UPPAAL needs a commercial licence for non-academic usage. Therefore its applicability in this workflow is limited and it is not fully integrated into the proposed verification workflow.

Pattern: $\mathcal{A}$ is always true at the end of the PLC cycle.
Formal representation (CTL): $\mathsf{AG}(EoC \rightarrow \mathcal{A})$

(a) Simple safety pattern

Pattern: If $\mathcal{A}$ is true at the end of the PLC cycle N and $\mathcal{B}$ is true at the end of PLC cycle $N+1$, then $\mathcal{C}$ is true at the end of PLC cycle $N+1$.
Tabular representation:

	Cycle N	Cycle $N+1$
Assume (at the end of the cycle)	$\mathcal{A}$	$\mathcal{B}$
Check (at the end of the cycle)	–	$\mathcal{C}$

Formal representation (LTL): $\mathsf{G}\left((EoC \wedge \mathcal{A} \wedge \mathsf{X}[(\neg EoC\ \mathsf{U}\ (EoC \wedge \mathcal{B})]) \rightarrow \mathsf{X}[\neg EoC\ \mathsf{U}\ (EoC \wedge \mathcal{C})]\right)$

(b) Complex safety pattern

Figure 3.3: Example requirement patterns

Verification model reductions. The models generated from the implementations are often large, it is difficult to avoid this problem. By making the generation smarter (and therefore more complex), the model size might be reduced, but this results in increased implementation effort and reduced maintainability. The model checkers often provide built-in reduction methods, but they cannot benefit from the domain-specific knowledge, i.e. the knowledge of the PLC's behaviour and the fact that the analysed models describe PLC programs. Including reductions in our verification workflow may reduce the severity of both issues: structural reductions can simplify the models without requiring complex translation rules, and domain-specific reductions may help to exploit the knowledge about the PLCs and the analysed programs. In addition, custom model reductions can help us to improve or fine-tune the model checking with relatively low effort, without developing a new model checker specifically adapted to the PLC domain.

This section discussed different aspects of the verification workflow. A workflow that follows the above discussed principles may be adequate for the defined needs. However, to put the different pieces together efficiently, a crucial part is missing: the intermediate representations. They will be discussed in the next section.

Publications related to this section. The steps of the proposed verification approach were first sketched up in [*r*24], then described in detail in [*c*16]. More details can be found in these publications.

3.3 Intermediate Representations

In order to *(i)* efficiently support multiple model checkers, *(ii)* to simplify the transformations between different artefacts and *(iii)* to help to include reductions in the verification workflow, the definition of intermediate representations is desired. This allows to make the verification workflow independent from the used model checker tool, as shown in Figure 3.4.

This section discusses the intermediate representations to be used in the workflow. The main focus is on the intermediate model (IM) language that can represent the model to be verified independently from the applied external model checkers and which also supports reductions (Section 3.3.1). Then

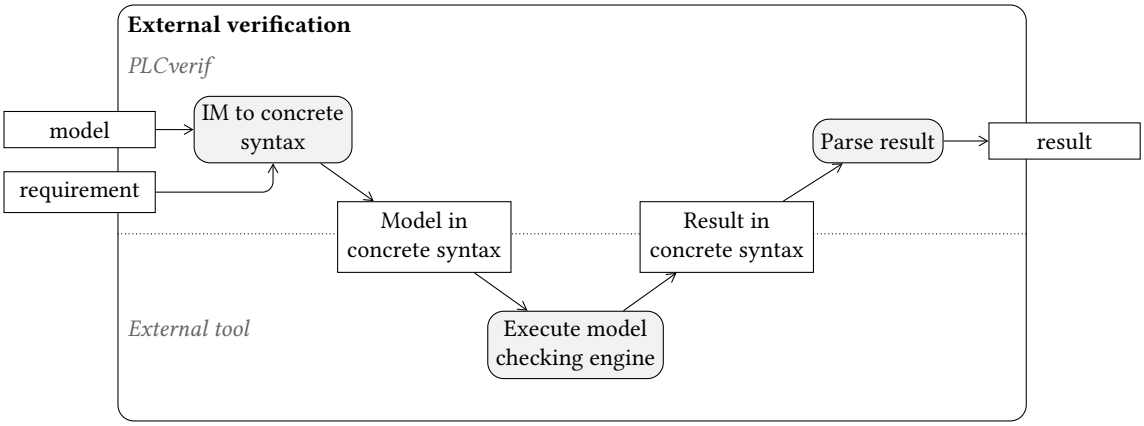


Figure 3.4: Wrapping the external model checking engine

the model checker-independent representation of the results and the counterexample is briefly shown (Section 3.3.2).

3.3.1 Intermediate Model: Intermediate Representation of the Verification Model

This section is devoted to the intermediate model (IM) language that can represent models of PLC programs.

Advantages of an intermediate verification model. Introducing a dedicated intermediate language to describe the verification models has various advantages.

- **Simplification of the transformation.** Creating a text-to-text transformation from the PLC code to the concrete syntax of a selected model checker is possible, however it tends to become very complex. By introducing the IM, the input and output languages are decoupled and the specialities of both the input and output languages can be handled independently.
- **Extensibility.** Besides simplifying the translation, the IM also eases the extension of the workflow. For example, to support the syntax of an additional model checker, it is enough to develop a translation from the IM, there is no need to handle the specificities of the PLC code again. Similarly, if a new PLC program language is added to the implementation, each supported model checker can be used without any modification.
- **Flexibility of the workflow.** By having an intermediate representation, it is easy to do manipulations on the model. For example, with the IM various model reductions can be applied before generating the concrete syntax for the model checkers. This way each supported model checker can benefit from the same reductions, the implementation of the reduction rules can be independent from the model checker used [*r*24].

Design aspects. Next, the constraints and requirements influencing the design of the IM language are discussed.

- **IM should capture the semantics, not the syntax.** The abstract syntax tree of the PLC program to be analysed is a syntactic description of the code. The elements of the syntax tree could not be mapped directly to the elements of generic verification models, as they describe

the semantics of the program using typically low-level formalisms. To facilitate the verification model generation, the IM should focus on the semantics of the programs, not the syntax.

- **IM close to the formalisms of the widely-used model checkers.** By designing an intermediate model language close to the targeted model checkers tools' input languages the simple "IM to model checker" translations can be ensured.

  Based on this design aspect, the IM language was designed based on the automaton theory. The IM describes a network of automata, extended with variables and simple synchronisations.

- **Restricted non-determinism.** The different tools handle non-determinism differently. Therefore non-determinism and undefined behaviours should be restricted in IM.

  Typically the PLC programs are deterministic, i.e. based on the current state of the program and the received inputs it can be determined what will be the next state of the program. In the IM each variable keeps its value unless it is explicitly modified by an assignment either to a defined value or to a non-deterministically chosen value. The IM allows to have multiple transitions enabled in different automata, but in the typical generated models this is only used to represent function calls, therefore there is no real concurrent behaviour in the models. Furthermore, we assume that in the generated models each location has at most one enabled outgoing transition, as the choice between multiple enabled outgoing transitions could be handled differently by the different model checkers.

- **Expressive enough to describe PLC programs.** An obvious requirement towards the IM is to be able to describe the behaviour of PLC programs. While it is simple to design an expressive language, it then has to be translated to the languages supported by the selected model checkers. Therefore IM was designed to cover a restricted set of PLC program features. We assume that the PLC program (SCL code) does not use pointers, nor any dynamic addressing, i.e. for each memory access the address can be determined in compilation time. This implies that for example arrays should not be addressed by variables (e.g. `arrayVariable1[var2]` is not permitted), except simple special cases (e.g. the index variable might be the iteration counter of a `FOR` loop). A further assumption is that PLC programs do not contain recursions.

  These assumptions are in accordance with the good practices of PLC programming and the development practices followed at CERN.

- **As simple as reasonably possible.** Programming languages typically have various constructs for usability, for the user's convenience. As the IM models will not be edited manually, it is not required to have such constructs. Therefore the IM is as simple as reasonably possible. For example, complex synchronisations, automata templates, variables with restricted scopes, arrays and structures are not supported in IM. The reduced feature set simplifies the reduction rules and the "IM to model checker" translations too.

  Moreover, contrarily to the modelling language used by UPPAAL, the IM language is not timed. The reasons for that are discussed later in this section.

It is interesting to note that the STL language (corresponding to IL in [I61131-3]) could also serve as an intermediate language, as each other PLC programming language could be translated to it [*c*11; *j*1]. However, this would be a "syntactic intermediate representation" instead of a semantic representation, e.g. it would contain function blocks, calls, etc. The semantics of these objects are relatively complex and not explicitly expressed in an STL-based intermediate model, while an automaton-based IM has

a simpler semantics, therefore it is easier to express in the concrete input languages of the various general-purpose model checkers. Other aspects of using STL as an intermediate formalism will be discussed later, in Section 3.6.2.

Timing aspects. Careful considerations were required about the timing aspects of the PLC program modelling. PLC programs may use pulse and delay timers (TON, TOF, TP in [I61131-3]). A straightforward idea would have been to design the IM such that it includes timing, e.g. by building it on the timed automata formalism. However, timed models would require timed model checkers. In the early experimentation phase of this project we have observed that this would significantly restrict the possible set of model checkers and would mean a serious drawback in performance. As timing is not a crucial aspect of our motivating examples, IM was designed to be non-timed.

The timing aspects are briefly discussed here. The purpose of this discussion is to justify the decision to use non-timed models. Precise descriptions and more detailed discussion can be found in [*c*14].

Modelling time of PLC programs was already targeted in various works, e.g. [MW99; Bel+10; WSG07; Wan+13]. Most of them use timed models ([MW99; Bel+10; WSG07]), but they do not present verification results for large PLC programs, therefore it is not known, how these modelling methods can cope with the state space explosion. [Wan+13] uses BIP models and represents time through a special signal which provides "ticks" with a fixed frequency. This method is less accurate than the timed automata models, but is still close to the real behaviour of PLCs.

In [*c*14], we proposed two timing representation approaches for the IM: the realistic and abstract modelling. Both representations are simplifications compared to the real behaviour of PLC programs.

The *realistic modelling* represents time with the same resolution as the real PLC hardware (typically 1 ms), but it assumes that the user code is executed in 0 time which is followed by a non-deterministically long delay. If the code contains one single PLC timer that is checked only once per cycle, this model provides the same result as the precise, real-life time representation would provide. If there are multiple timers, certain corner case behaviours are lost, but this is typically not problematic [*c*14]. The advantage of this simplification is that there is no need for timed models or model checkers supporting timing, the timers and the clock can be represented using integer variables, as discussed in [*c*14].

Contrarily, the *abstract modelling* does not model the time precisely, to reduce the state space of the models to be verified. The abstract modelling uses an abstract, non-deterministic representation of the PLC timers.

The main difference between the two representations can be demonstrated by two example requirements. Using the realistic modelling it can be evaluated whether "a certain PLC output is set to true 500 ms after a given input has been set to true". The abstract model cannot provide a precise answer to this question, but it can decide whether "a certain PLC output will be set to true (eventually) after a given input is set to false" [*c*14].

High-level syntax and semantics of the intermediate model. The high-level syntax and semantics of the IM was introduced in [*j*3] as follows. It has to be noted that the IM language is similar to the modelling language used in UPPAAL [Amn+01], but IM is kept much simpler, according to the design aspects discussed before. This way the IM language is close to the input language of UPPAAL, but also NuSMV/nuXmv and any other model checker that uses variants of automata or state machines. Furthermore, the IM is close to the control flow graph (or control flow automaton) representation of the PLC programs.

Definition 3.1 (Intermediate model language, based on [*j*3]). The intermediate model language is defined as a simple automata network model consisting of synchronised automata.

A *network of automata* is a tuple $N = \langle A, I \rangle$, where A is a finite set of automata, I is a finite set of synchronisations.

An *automaton* is a structure $a = \langle L, T, \ell_0, V_a, \underline{Val}_0 \rangle \in A$, where $L = \{\ell_0, \ell_1, \dots\}$ is a finite set of locations, T is a finite set of guarded transitions, $\ell_0 \in L$ is the initial location of the automaton, $V_a = \{v_1, \dots, v_m\}$ is a finite set of variables, and $\underline{Val}_0 = (Val_{1,0}, \dots, Val_{m,0})$ is the initial value of the variables.

Let $\hat{V}$ be the set of all variables in the network of automata N, i.e. $\hat{V} = \bigcup_{a \in A} V_a$. ($\forall a, b \in A : V_a \cap V_b = \emptyset$ if $a \neq b$)

A *transition* is a tuple $t = \langle l, g, amt, i, l' \rangle$, where $l \in L$ is the source location, g is a logical expression on variables of $\hat{V}$ that is the guard, amt is the memory change (variable assignment, i.e. a function that defines the new values of the variables in $\hat{V}$), $i \in I \cup \{NONE\}$ is a synchronisation attached to the transition, and $l' \in L$ is the target location. amt may assign new values to multiple variables, but the order of their evaluation is undefined.

A *synchronisation* is a pair $i = \langle t, t' \rangle$, where $t \in T$ and $t' \in T'$ are two synchronised transitions in different automata. The variable assignments attached to the transitions t and t' should not use (assign or refer to) the same variables. This composition operation is restrictive, but sufficient to model PLC programs, as synchronisations will only represent function calls. Synchronisations are often used only to simplify the initial model generation and are later removed by reductions that merge the different automata, making the synchronisations unnecessary.

The operational semantics of this automata-based formalism can be informally explained as follows: a transition $t = \langle l, g, amt, i, l' \rangle$ from the current location l of an automaton is enabled if g is satisfied and either t has no synchronisation attached, i.e. $i = NONE$, or $i = (t, t')$ and the transition t' is also enabled. In the former case, t can fire alone; in the latter case, both t and t' have to fire simultaneously. Each execution step consists in firing one transition or simultaneous firing of two synchronised ones. Upon firing of a transition t as above, l' becomes the new current location of the corresponding automaton and the new values of variables $\hat{V}$ are set using the previous values and the variable assignment amt. [*j*3] ▪

A formal semantics definition based on systematic translation to a state-transition system can be read in [*r*23]. The detailed discussion of the metamodel of the intermediate model language is in Appendix C.1 (p. 131).

Translation from PLC code to IM. The translation from SCL code to IM is described as a systematic, rule-based recursive translation of each element in SCL code to the intermediate model. As this is not a contribution of the current dissertation, the details are not discussed here. Interested readers can find more information and examples about the translation from SCL code to IM in [*r*24; *j*3]. [*r*24; *r*23] also describe the main rules of the IM to NuSMV model translation.

The model checking workflow and the PLCverif tool implementing it provide solutions to the discussed challenges and issues. They provide a complete workflow that is automated and hidden from the user. Based on the source code and the filled requirement patterns, the model checking is automatically performed and a verification report is produced [*c*8].

3.3.2 Additional Intermediate Representations

In order to make the verification workflow independent from the chosen model checker tool and to make is easily extensible, both the inputs and outputs of the model checker tools should be hidden (as shown in Figure 3.4). Therefore the results provided by the model checker have to be parsed and translated into an intermediate, tool-independent format.

Furthermore, providing the intermediate model is not enough: a mapping has to be established between the variables of the PLC program, the variables of the intermediate model and the variables of the model checkers' concrete syntax. This permits to provide the results in a format that is understandable for the users, i.e. using the names defined in the PLC code.

The metamodel of the output and counterexample representation and the mappings between the various representations of the variables are described in Appendix C.2 (p. 133).

Publications related to this section. The design of IM was first discussed in the technical report [*r*24]. This report provided also the first description of the syntax and semantics of IM, which was later refined in [*r*23; *j*3].

3.4 Verification Workflow Based on the Intermediate Model

After discussing the details of the IM in Section 3.3, the verification procedure described in Section 3.2 can be concretised. A simplified overview of the procedure can be seen in Figure 3.5.

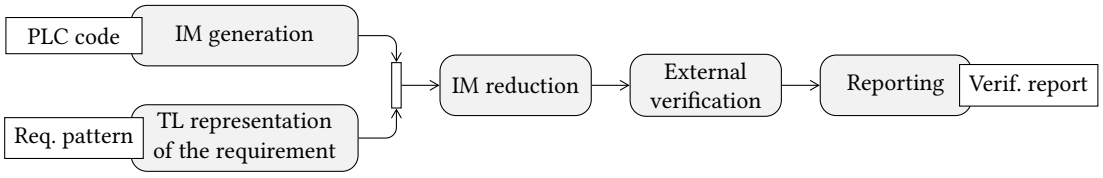


Figure 3.5: Simplified overview of the PLC program verification workflow

The complete workflow is discussed and summarised in [*j*3]. The workflow is presented from the users' point of view in the following.

1. The workflow is based on the *PLC program* and a requirement, formalised using one of the *requirement patterns*.
2. An *IM representation* is generated from the PLC program. The main translation rules for SCL are described formally in [*j*3].
3. The requirement pattern is automatically transformed into a CTL or LTL *temporal logic (TL) representation*.
4. The intermediate model is then *reduced* (see the details in Section 3.5).
5. The reduced IM and the requirement are then translated to the concrete syntax of the external model checker tool. The *model checker is executed* and its result is parsed, representing it in a model checker-independent format.
6. The result of the procedure for the user is a *verification report* that describes all details and the result of the verification.

The extensibility of the approach was shown in [*j*3], where initial work on extending the workflow to model interrupts, to handle SFC inputs and to cope with large, complete PLC applications were presented. Extensions to support the verification of safety-critical PLC programs (and therefore to support the STL language) are discussed in Section 3.6.

Publications related to this section. The steps of the proposed verification approach were first described in detail in [*c*16]. Detailed overview of the verification workflow can be found in [*j*3]. The steps are presented from the users' point of view in [*c*13].

3.5 Reduction Rules for the Intermediate Model[4]

The previously presented intermediate model language is able to represent the behaviour of the PLC programs for verification purposes. Furthermore, it is possible to generate a concrete representation of this model for a chosen model checker. However, our early experiments have shown that the enormous size of the generated models undermine the possibility of their verification. Even though certain model checkers (e.g. NuSMV/nuXmv) contain built-in reduction techniques, they cannot cope with most of these generated models.

The intermediate model representation permits the integration of various reduction techniques into the verification workflow. Three main categories of reductions are presented here: *(i) mode selection* which permits to focus efficiently on certain scenarios only (Section 3.5.1), *(ii)* a *cone of influence* heuristic which removes variables that are not required for the evaluation of the given requirement (Section 3.5.2), and *(iii) rule-based, structural reductions* which can simplify and reduce the size of the IM (Section 3.5.3). All of the presented reductions are heuristics: they do not aim to make the IM size optimal (minimal), but to make the IM smaller with relatively low resource needs.

General requirements towards the reductions. Before presenting the details of each reduction category, some common designs aspects are discussed.

- **The satisfaction of the requirement should not be altered.** The reductions should not alter the result of the requirement evaluation, thus we are targeting property-preserving reduction techniques. We assume that no requirement refers the value of any variables in transient states, i.e. they should only be checked at the beginning or at the end of the PLC cycle. This is a valid assumption, because the transient signals are not observable outside of the PLC. It is also assumed that there is no concurrency in the PLC programs, as PLC programs are typically single-threaded and without interrupts. This work is not targeting the analysis of concurrency problems.
- **Simple reduction rules.** The complex reduction rules might be error-prone, therefore most reduction rules are simple, they are performing single reduction steps. This implies that the application of a reduction rule may enable another reduction rule. Therefore most of the reductions should be applied iteratively.
- **Finite reduction loops.** In case of iterative reductions it is a primary requirement to have finite reduction loops only, the reduction rules should not enable each other infinitely.
- **Do no harm.** The precise reductions are often highly resource-consuming. Aiming for an optimal model typically needs a lot of resources, but if the reduction consumes more than the

[4]Both this work and [Fer14] describe reduction rules and build on the work [*c*15]. The high-level idea of the verification workflow is a joint contribution with B. Fernández. The variable abstraction reduction method (not presented here) is the contribution of B. Fernández, while the detailed design and formalisation of the other reductions are the author's contributions.

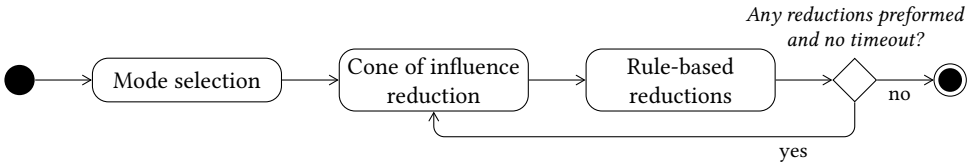


Figure 3.6: Iterative reduction workflow (based on [c15])

gain in verification time, the overall effect of the reductions is negative. Therefore the reduction heuristics are kept simple and lightweight, even if this means that they cannot cope with certain scenarios when the model could be further reduced.

The iterative reduction loop is shown in Figure 3.6. The mode selection is independent from the rest of the reductions, therefore this does not have to be applied iteratively. The rule-based reductions and the cone of influence reduction are applied until a given exit condition is satisfied. If the last iteration of the reduction loop did not modify the model, the reduction loop can be terminated. The loop can also be terminated if it was running for too long time or if the model is already small enough.

In the following the three mentioned groups of reductions will be discussed.

3.5.1 Mode Selection

The goal of the mode selection is to allow the user to fine-tune the verification by setting an operation mode to be verified. For example, some requirements need only to be satisfied in certain modes if certain inputs or parameters have given values.

The motivation of this reduction method is coming from UNICOS (Unified Industrial Control System), a CERN PLC framework that contains a set of basic, generic objects[5] (so-called baseline objects). These objects can be adapted to the specific usage by setting their parameters which are special input variables that do not change their values during execution [c15].

If a requirement ϕ shall be satisfied only if a parameter p is true, this can be expressed easily using a CTL expression: $\mathsf{AG}(p) \rightarrow \mathsf{AG}(EoC \rightarrow \phi)$, where EoC is an atomic proposition that is true only at the end of each PLC cycle. Certain model checkers (e.g. NuSMV/nuXmv) permit to define invariants in the model, $p = \textit{true}$ could be expressed in this way too. However, our initial experiments demonstrated that better performance can be achieved if these simple invariants are incorporated in the model itself. As the verification workflow relies on the intermediate model, this can be achieved easily.

The mode selection reduction gets an input $v_1 = c_1, v_2 = c_2, \ldots$, where each v_i is a variable and c_i is a constant. Then in the IM the mode selection algorithm replaces every occurrence of v_i with c_i for every i. After this reduction, the requirement can be simplified: e.g. in the above example the $p = \textit{true}$ assumption does not have to be included, and the requirement to be checked by the model checker can be simplified to $\mathsf{AG}(EoC \rightarrow \phi)$.

The application of mode selection has several advantages compared to including these assumptions about the mode of the verified PLC program in the requirement (based on [c15]).

- The state space of the model is reduced as mode selection eliminates the variables v_i.
- The application of mode selection can help the rest of the reductions. For example, if v_1 is set to false, later a rule-based reduction may replace the expression $v_1 \wedge w_2 \wedge w_3$ with constant false.

[5] More details about UNICOS are discussed in Section 3.8.

- As mode selection may cause the simplification of expressions, this can also help the cone of influence reduction which removes the unused variables. In the previous example, if w_2 and w_3 are not used elsewhere in the model, they can be removed by the cone of influence reduction.

3.5.2 Cone of Influence

Program slicing [Wei81] or cone of influence reduction (COI) is a widely-used abstraction technique. Its motivation is that certain requirements can be evaluated by analysing only a part of the model. For example, if a PLC program produces the outputs $o_1, o_2, \ldots, o_n$, but in the requirement only the value of o_1 is used and the other output variables do not influence o_1, these other output variables are not necessary for the verification. These variables and the computation of their values can be removed from the model. This reduction may make other variables unnecessary, thus in total the effect of the reduction is often very significant.

Program slicing is often used in software verification, e.g. in [LNN13; BMP15; KSK15], on the control flow graph of the program under verification. Certain model checkers, e.g. NuSMV/nuXmv also contain built-in cone of influence reductions. However, as their input model is on a lower abstraction level than the IM, this reduction is less efficient [*c*15]. This motivated the design and implementation of a custom, heuristic COI algorithm to be included in PLCverif.

This COI algorithm builds a variable dependency graph. In this graph the nodes are the variables defined in the IM. The directed edges represent dependencies between variables. There are two types of edges, representing assignment or data dependencies (e.g. $v_1 := \neg v_2$ will imply a data dependency edge $v_1 \rightarrow v_2$) and guard or control dependencies (e.g. if v_1 is assigned by a transition having a guard $[v_3]$, it implies a control dependency edge $v_1 \rightarrow v_3$).

As it was discussed before, it is not necessary to provide an optimal solution. This custom COI is a heuristic algorithm that only removes variables and variable assignments that will not influence the result, but it does not necessarily remove all of them.

The heuristic identification of data dependencies is done by checking each variable assignment $\langle v := \mathit{Expr} \rangle$ in the IM. The identification of control dependencies in an arbitrary IM model is more difficult[6]. The heuristic is based on the identification of *unconditional locations* of the IM. These are the locations that will be "visited" in each PLC cycle independently from the input values or computations in previous cycles. Typically, the conditional branches are small in the PLC programs, and therefore there are many unconditional locations in an IM. It is easy to see that if a variable v is assigned on an unguarded transition t leaving an unconditional location will not imply any control dependency, if t does not have any guard. For assignments of v on transitions leaving conditional locations, the guards are collected backwards until the first unconditional locations and v will be considered as depending on each of the collected variables. Note that this might be an over-approximation.

The formal discussion of this COI algorithm can be found in [*c*15]. It is worth to be noted that by building and using the dominator trees [All70] of the intermediate model the precision of the heuristic might be improved. There are various more aggressive program slicing methods (e.g. [KSK15]) which may achieve greater reductions. The improvement of the current COI heuristic is a future work.

[6]This is the main difference between the custom and NuSMV's COI: NuSMV cannot identify the control dependencies precisely, as the NuSMV model does not distinguish between variables and the structure of the automaton. More detailed analysis of the differences between the custom and NuSMV's COI can be found in [*c*15].

3.5.3 Rule-Based Reductions

This part is dedicated to the rule-based reductions. This is a diverse group with various reductions, ranging from the very simple ones (e.g. replacement of the "$\cdots \wedge F \wedge * \wedge \cdots$" expressions with constant "F") to more complex ones. These reductions are similar to the ones used in compilers to optimise the control flows [CT12].

These rule-based reductions aim to simplify the structure of the IM or reduce the number of variables (variable assignments). The rules can be divided into three main categories:

- *Automaton simplifications.* These reductions eliminate unnecessary transitions and locations. For example, the reduction illustrated[7] in Figure 3.7(a) removes empty conditional branches, the reduction in Figure 3.7(b) eliminates the transitions without any described behaviour. Some other reductions (e.g. the one in Figure 3.7(c)) reduces the number of locations and transitions by attaching multiple, independent variable assignments to the same transition.
- *Variable simplifications.* These reductions analyse the data flow of the IM and try to make improvements based on this. For example, if two variables have always the same value at the end of any PLC cycle, certain reductions try to merge these two variables.
- *Expression simplifications.* A group of reductions is responsible for the simplification of Boolean expressions, by reordering them, identifying if an expression or a subexpression is constant or if it contains unnecessary elements. It has to be noted that some reductions have opposite effect: they build more complex Boolean expressions, for example to merge several guard conditions on successive transitions into one (e.g. the reduction shown in Figure 3.7(d)).

A part of these reductions are general-purpose, domain-independent: e.g. the expression "$v_1 \wedge F$" can always be simplified to "F", if it has no side effects. Certain reductions are domain-specific, and they benefit from the assumption that the transient states are not checked, e.g. two variable assignments can be reordered if it does not influence the output at the end of the PLC cycle.

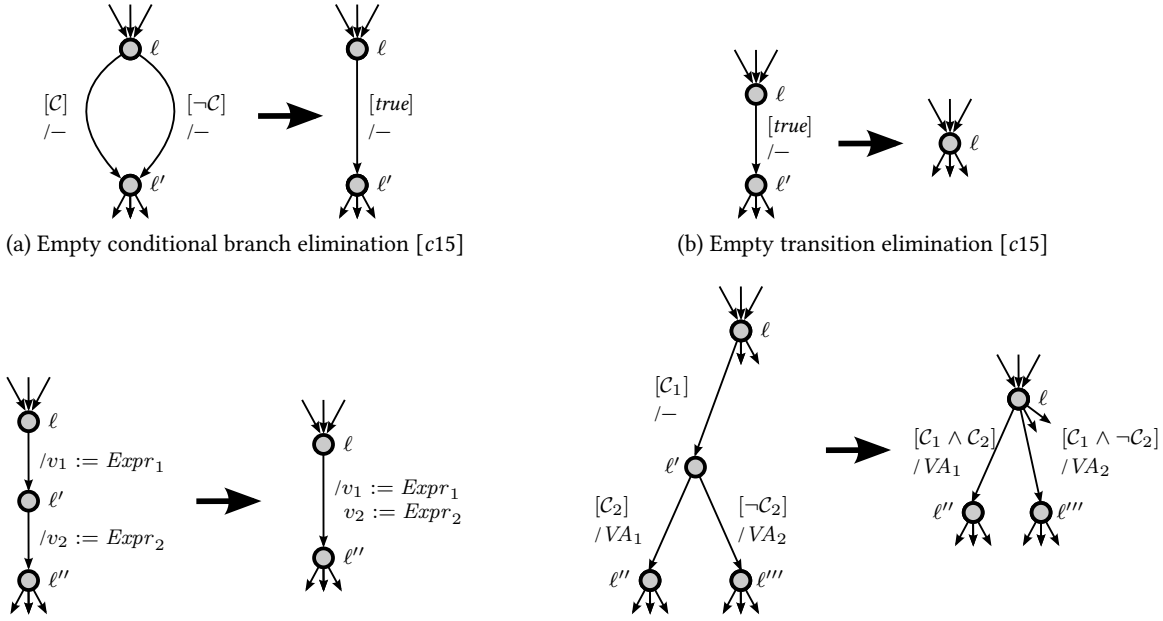
3.5.4 Reduction Examples

This part presents simple examples of the reduction methods considered above. Let us consider the SCL program in Listing 3.1 with the following requirement: *"If the parameter $pInc$ is set to 1, the variable c cannot be negative."* The following examples are extended versions of the examples presented in [c15].

- Figure 3.8(a) shows the IM corresponding to the example SCL program in Listing 3.1.
- First the mode selection is applied. The requirement above targets only the cases when $pInc = 1$. The IM after the mode selection can be seen in Figure 3.8(b).
- The next step is the COI reduction. The variable dependency graph is shown in Figure 3.9. The data dependencies are represented with thin red, the control dependencies with thick blue edges. Based on the variable dependency graph it is obvious that for a requirement targeting only the variable c, no other variable is required. Therefore the rest of the variables will be removed by COI, as shown in Figure 3.8(c) and Figure 3.8(d).
- After the COI reduction, the rule-based structural reductions can reduce the IM by removing unnecessary locations and transitions from the model. By applying the reductions shown in Figure 3.7(a) and Figure 3.7(b), the final IM is the one presented in Figure 3.8(e).

The efficiency of reductions is demonstrated on larger, real-life examples in Section 3.8.

[7] The example IM snippets in the following denote locations with circles, transitions with arrows, guards with square brackets, and variable assignments preceded by "/".



(a) Empty conditional branch elimination [c15]

(b) Empty transition elimination [c15]

(c) Variable assignment merging rule [c15]

(d) Condition push-down rule

Figure 3.7: Example structural reduction rules

Publications related to this section. The usage of model reduction heuristics was first mentioned in the technical report [r24] as a motivation for the usage of IM. The case study in [c16] already benefited from the reductions and the effects of different reduction types were discussed. Detailed discussion of the various reduction techniques were provided in [c15], then later they were summarised in [j3].

3.6 Extensions for the Verification of Safety-Critical PLC Programs[8]

PLC-based systems are more and more accepted and used in safety-critical settings [Gre94]. In these cases the logical faults introduced by mistake, misunderstanding or oversight can cause potentially dangerous situations (among other reasons) [Par03].

Originally, the previously described verification workflow targeted mainly programs written in SCL language. To support programs written for safety PLCs (also called safety-critical or fail-safe PLCs), other programming languages have to be supported as well. This section discusses the challenges related to the verification of safety PLC programs and the provided solutions.

3.6.1 Motivation and Challenges

The safety-critical controllers have to fulfil the requirements of the corresponding standards, such as IEC 61508, IEC 61511, or IEC 62061. These standards define different *safety integrity levels* (SIL) and various requirements and guidelines for the system and the development process. Many PLC vendors produce a special range of hardware complying with the corresponding standards. These so-called fail-safe PLC CPUs (or simply *safety PLCs* in the following) are typically certified up to SIL3 according to IEC 61508-2 [I61508-2]. Besides the special hardware, the PLC vendors provide special development environments, often with additional restrictions compared to non-safety-critical PLC

[8] This section is an extended and adapted version of [c9].

```
1  FUNCTION_BLOCK FB
2      VAR_INPUT
3          ia : INT;
4          ib : INT;
5          pInc : INT;
6      END_VAR
7      VAR_OUTPUT
8          xa : BOOL;
9          xb : BOOL;
10         c : INT := 0;
11     END_VAR
12 BEGIN
13     IF ia > 0 THEN
14         xa := TRUE;
15     ELSE
16         xa := FALSE;
17         IF ib > 0 THEN
18             xb := TRUE;
19         ELSE
20             xb := FALSE;
21         END_IF;
22     END_IF;
23
24     c := c + pInc;
25 END_FUNCTION_BLOCK
```

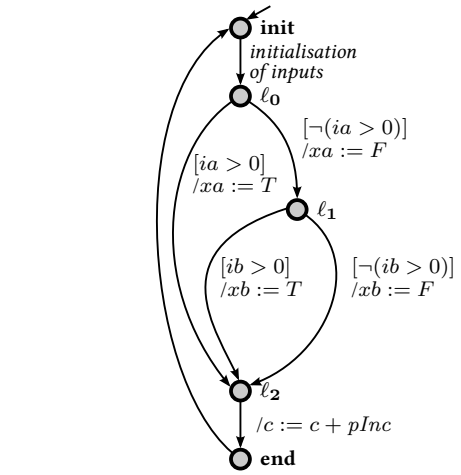
Listing 3.1: SCL code used in the reduction example (based on [*c*15])

programming. For instance, Siemens restricts the developers to use the LAD or FBD language with further restrictions, such as no usage of floating-point or compound data types [Sie14], following the recommendations of the IEC 61511-2 standard [I61511-2]. Although the hardware of the safety PLCs is special, the hardware differences do not affect the software part. Thus the main particularity of the safety PLCs for the verification is the restricted programming possibilities, namely the obligation to use restricted LAD or FBD language for programming (in case of Siemens PLCs) [*c*9].

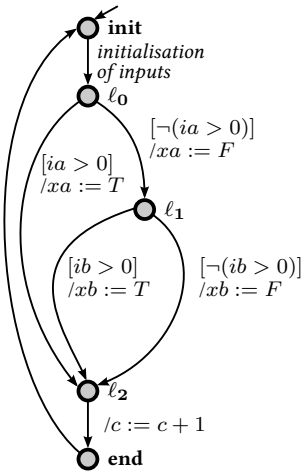
One of the motivation of the previously presented verification workflow was the reduced amount of verification resources in non-safety-critical cases, which makes the methods requiring deep expertise impossible. Obviously, more effort is spent on the verification of safety-critical systems, however using formal verification for safety PLCs is still not common. The IEC 61508-3 [I61508-3, Table A.4] marks the usage of formal methods "highly recommended" for the detailed design and development only for SIL4 (the highest SIL defined), and even there it can be replaced by "structured methods" or "semi-formal methods". This is in accordance with our observations at CERN. Probably one of the most critical PLC-based systems at CERN is the access control system of the accelerator complex. It is responsible for personal safety by not letting people enter areas where the accelerator is in use, thus there is a risk of dangerous radiation levels and other hazards. The state-of-the-practice for verification of this system is the extensive usage of testing [Val+08; Val+13].

Challenges. There are two main challenges related to the verification of safety-critical PLC programs (written for Siemens safety PLCs).

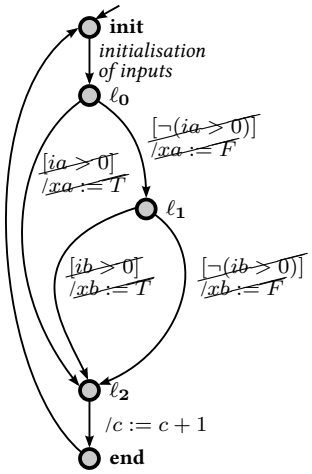
- The primary need to verify safety-critical PLC programs is the ability to check LAD and FBD code. However, in case of Siemens PLCs, the programs written in graphical languages are not directly accessible, but they can be exported from the development environment as STL code.



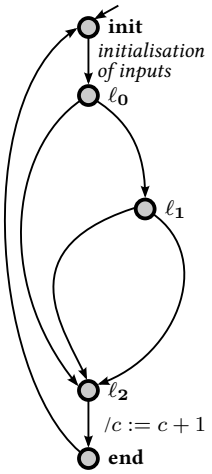
(a) IM corresponding to the example SCL code



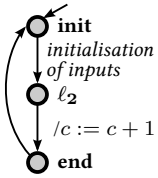
(b) IM after mode selection



(c) IM after COI



(d) IM after COI



(e) IM after COI and structural reductions

Figure 3.8: Example IM representations in different stages of reductions (based on [c15])

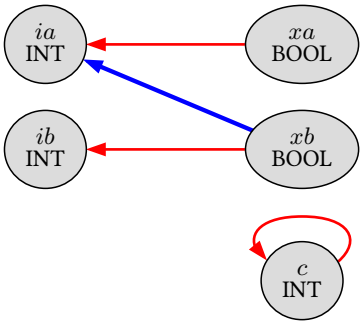


Figure 3.9: Variable dependency graph of the example SCL program (after mode selection)

Therefore the challenge is to support simple STL programs, mainly having Boolean and arithmetic operations (as these are the typical blocks used in safety logics). The main challenge of supporting STL is the lack of precise semantics definition.
- The STL instructions use low-level data structures and have many side effects. This may result in large models. It is therefore required to extend the verification workflow with new reductions that target specifically the IMs generated from STL programs.

3.6.2 Supporting the STL Language as Input Language

As discussed before, the STL language has to be supported by the verification workflow in order to verify safety-critical PLC programs. However, the abstraction level of STL is low, therefore handling STL programs is different from handling SCL programs.

It is resource demanding to implement *(i)* a parser that can build the abstract syntax tree of the STL language, and *(ii)* a model translator that translates the syntax tree to the intermediate model. Therefore we tried to find a pivot language that can represent PLC programs written in any of the Siemens PLC languages. At first look STL – being a low-level language – might even seem to be a good pivot (intermediate) language, as the SCL programs could be translated to STL. Earlier work [*c*11] discussed that STL may represent programs written in any of the five programming languages used in Siemens PLCs. However, it was also shown that the SCL language extended with explicit register representation can also be used as a pivot language. SCL is a higher-level language, with a more compact representation (e.g. for expressions). The IM, used under the verification workflow also supports complex expressions, similarly to many model checkers, e.g. nuXmv, UPPAAL. Hence translating a compact SCL expression to a lengthy STL form seems to be inefficient. Furthermore, as in our setting mostly SCL programs are verified, using SCL (and not STL) as pivot can provide support for the other languages without any impact on the verification of SCL programs [*c*11].

Consequently in the following the translation from STL to SCL will be discussed[9]. This translation provides a mapping in an inductive way from each STL statement to (one or more) SCL statements. This way the method of IM construction does not change, and the instruction-by-instruction mapping can be much simpler than a complete parser for the STL language. In [*c*11; *j*1] detailed discussions about the possibility of representing STL programs in SCL can be found.

3.6.2.1 Mapping STL to SCL

The challenge of this instruction-by-instruction mapping is that the STL instructions directly access and modify the low-level data structures of the PLC (e.g. registers). For example, the STL statement `L var1` stores the contents of Accumulator 1 in Accumulator 2, then it loads the value of variable `var1` to Accumulator 1. There is no language element to access the registers directly in SCL, making the direct representation of STL code impossible. However, this can easily be solved for verification purposes. We emulate the registers as local SCL variables according to a well-defined naming convention, and use it consistently in the SCL programs and in the properties to be verified. To avoid the confusion – though it does not require a language extension –, we will use *SCLr* as language name for programs written in SCL where the registers are emulated as local variables. This solution is similar to the one presented in [SD08]. To distinguish between ordinary variables and the ones representing SCLr registers, the latter's names start with double underscores.

[9] It has to be noted that from the theoretical point of view this does not cause any additional problems, the same challenges would have been targeted if the verification of STL programs was supported from the beginning, as can be read later on.

Table 3.1: STL to SCLr transformation examples [*j*1]

IL instr.	SCLr equivalent
`A var1`	`IF __NFC THEN __RLO:=__RLO AND (var1 OR __OR); ELSE __RLO:=var1 OR __OR; END_IF;` `__STA:=var1; __NFC:=TRUE;`

`A(`	`__nsRLO[8]:=__nsRLO[7]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;`
	`__nsOR[8] := __nsOR[7]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=__OR AND __NFC;`
	`__nsFC2[8]:=__nsFC2[7]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[8]:=__nsFC1[7]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[8]:=__nsFC0[7]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

`>I`	`__OR:=FALSE; __NFC:=TRUE;` `__RLO:=(__ACCU1<__ACCU2); __CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);`
`L var1`	`__ACCU2 := __ACCU1; __ACCU1 := var1;`

With this extension, each STL instruction (e.g. bit logic and comparison operations, conversions, jumps, arithmetic instructions, load and transfer instructions) can be represented in SCLr, by making all implicit effects of the STL instructions explicit in SCLr. For this purpose, we have identified the semantics of each STL instruction by checking on real PLCs what the results of the instruction for every possible initial state are (i.e. for each valuation of the read registers and variables). The identified semantics of the STL instructions are generic, not specific to our case studies. Some examples of this translation with different complexities are in Table 3.1. A short description of the used registers is in Table D.1 (p. 136).

As each STL statement can be translated into SCLr, it can be seen inductively that each STL program can be translated into SCLr as well. In other words, SCLr can emulate all STL programs, and consequently all FBD and LD programs too.

Furthermore, it is worth to note that SCLr can be regarded as a textual concrete syntax of the PLCverif intermediate model, therefore there is no theoretical difference if we translate STL programs to the intermediate model directly or through SCLr; translating STL through SCLr or directly to IM does not impose any theoretical difference. The main challenge is the same in both cases: determining the semantics of STL, which is targeted in Section 3.6.2.2.

A more detailed discussion of the memory model of STL and the registers can be found in [*j*1].

3.6.2.2 Determining the Semantics of STL[10]

Certain documentations are provided by Siemens about the semantics of the STL language. However, these are informal, incomplete descriptions. Often it is not possible to determine the precise semantics of each STL instruction (e.g. which variables and registers they modify and rely on).

In order to discover the precise semantics, we execute the instructions in all possible combinations, i.e. checking the results of each instruction in each possible situation. Of course, testing the behaviour of each instruction with each possible memory content is not feasible. However, each instruction depends only on certain registers and certain parts of the memory. These dependencies are defined in the description or status word influence part of [Sie98a; Sie10]. It is also defined (or can be assumed based on the description), which registers and memory locations might be altered by the execution of a certain instruction[11]. Reproducing all possible combinations of the registers and parameters that

[10] This section is an extended version of the Section 6 of [*j*1].

[11] It is precisely defined which status bits can be modified by the instruction, but the same information is not given for other registers or memory locations.

```
1  L a
2  L b
3  >=I
4  = r
```

(a) Source STL code

```
1  r := (a >= b);
```

(b) Equivalent SCL code

```
1  __ACCU2 := __ACCU1;  __ACCU1 := a;
2
3  __ACCU2 := __ACCU1;  __ACCU1 := b;
4
5  __OR := FALSE;  __NFC := TRUE;
6  __RLO := (__ACCU1 <= __ACCU2);
7  __CC0 := (__ACCU1  > __ACCU2);
8  __CC1 := (__ACCU1  < __ACCU2);
9
10 IF __MCR THEN r := __RLO; END_IF;
11 __OR := FALSE;  __STA := r;  __NFC := FALSE;
```

(c) Generated SCLr code

Figure 3.10: Illustration of code blow-up caused by STL-to-SCLr translation [*j*1]

affect a specific instruction and checking the new values of the altered affected registers and memory locations is feasible [*j*1]. On certain PLCs it is possible to modify the value of the status word. It is also possible to determine the current values of the status bits and registers on a breakpoint using a real PLC or the official Siemens PLC simulator software. Therefore we can generate test programs which execute a given instruction for each "interesting" valuation.

An example is given in Appendix D.1 (p. 135) showing the steps of determining the semantics of a STL instruction. This example demonstrates also that there may be discrepancies between the intuitive meaning of the informal descriptions (defined e.g. in [Sie02]) and the observed behaviour, thus the STL-to-SCLr translation cannot rely only on the informal description of the instructions.

The proposed approach to demonstrate the correctness of this STL to SCL transformation is summarised in Appendix D.3 (p. 137) based on [*j*1]. Three steps are performed to show the semantic correspondence: *(i)* the formal semantics of SCL is drawn up, *(ii)* the formal semantics of base STL (e.g. variable access, sequence of statements; but the semantics of the statements is omitted) is sketched up, then *(iii)* a proof strategy is given that shows that the SCLr equivalent of the STL instructions will cause exactly the same memory modifications as the STL statements in the systematic observations. For this last step, each observation table is formalised using the base STL semantics row by row.

3.6.3 Code Size Blow-Up and Reductions

Representing the registers as local variables allows the inductive mapping of STL programs to the SCL language, allowing to reuse the PLCverif workflow and toolchain. However, it raises a new concern: a single STL instruction may read and modify several registers. This causes a significant code size blow-up, as illustrated in Figure 3.10. The original sample STL code contains 4 instructions (Figure 3.10(a)), which can be represented by one single statement in SCL (Figure 3.10(b)). However, the STL code translated to SCLr has 13 variable assignments (Figure 3.10(c)). Note that these assignments represent the storage of (intermediate) results that are not necessarily needed by the subsequent statements. The extremities of this are the nesting Boolean operators (e.g. `A(`). They store some intermediate computation results in the so-called *nesting stack*, helping the developers to handle complex Boolean operations. Therefore a single STL statement might be translated to 40–50 SCLr assignments (see instruction `A(` for example in Table 3.1 that is represented using 43 SCLr statements).

This blow-up effect can be reduced by developing new automated reduction heuristics, similarly to the ones already included in the verification workflow described in the previous sections.

- *Expression propagation* can help to reduce the number of assignments. For example, the second assignment of line 1 in Figure 3.10(c) can be removed and the first assignment of line 3 can be

replaced by `__ACCU2 := a;` without modifying the behaviour of the program, i.e. without modifying the variable values at the end of any PLC cycle.
- *Assignments without any observable effect* can be removed. For example, the first assignment of line 1 in Figure 3.10(c) can be removed, as its effect is hidden by the first assignment of line 3.
- The non-used variables are deleted by the already existing *cone of influence* reduction. For example, the `__CC0` and `__CC1` register-representing variables can be removed, as they are never read in Figure 3.10(c).
- The expression propagation can result in complex Boolean expressions, which can be reduced by *Boolean factoring* and other *Boolean expression reduction methods*. If the simplified expression refers to fewer variables, these reductions may help the cone of influence reduction. Nevertheless, even if they do not reduce the state space, the Boolean expression simplification makes the other reductions faster and decreases the memory needs.
- A new *element-of* relation was also introduced in the IM language. This makes the representation of the recurring `a = 1 OR a = 2 OR a = 5 OR ...` pattern more efficient by representing as `a ∈ {1, 2, 5, ...}`.

By using these reduction heuristics, the code in Figure 3.10(c) can be automatically reduced to the one in Figure 3.10(b), assuming that the registers are not read by any further part of the code. Note that each reduction is applied only if it preserves the satisfaction of the requirement that is currently under evaluation.

Publications related to this section. This section is mainly based on [*c*9] that presented the first safety-critical verification case study using our procedure. Details about the semantics and the transformation of the STL language are published in [*c*11; *j*1].

3.7 Implementation[12]

The presented verification workflow was implemented in the PLCverif tool. In this section we overview the main features of the PLCverif, focusing on the user's point of view. The structure of this section follows the normal workflow of a user.

The typical user workflow of the PLCverif tool consists of four steps:
1. Defining (importing or writing) the PLC code to be checked,
2. Defining the requirement to be verified using requirement patterns,
3. Executing the verification (including IM generation, reductions and the execution of the selected external model checker), and
4. Evaluating the results of the verification based on the provided verification report.

In the following paragraphs, each step is described in detail.

Defining the PLC code. The PLCverif tool provides an editor for PLC programs. Currently the tool mainly supports SCL programs. Additionally, programs in SFC and STL are partially supported through SCL, with automated conversion. The code editor of PLCverif (Figure 3.11) provides the main features required nowadays in modern development tools, e.g. syntax highlighting, content assist, support for refactoring. The PLC program to be verified can either be written in this PLC code editor, or imported if the program already exists.

[12] This section is an extended and adapted version of [*c*13].

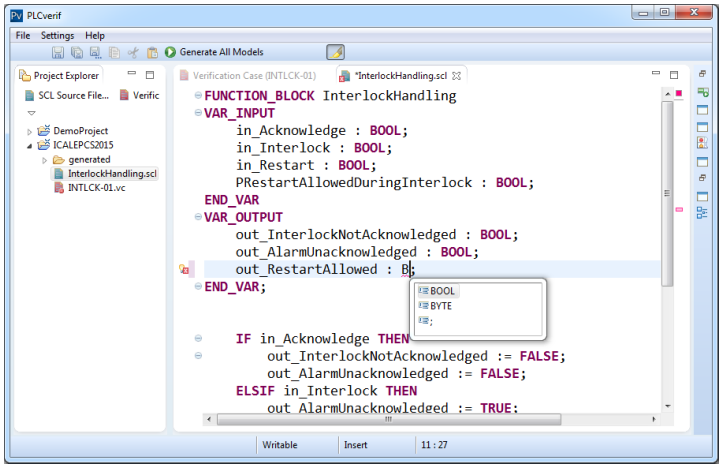


Figure 3.11: PLC program editor PLCverif [*c*13]

Defining the requirement. Similarly to test cases in testing, a *verification case* should be defined by the user that contains all necessary information for the verification. The user has to:

- Provide the *general information* of the verification case (ID, name, description),
- Select the *source code* to be checked,
- Define the *requirement* to be checked, and
- Select the external *model checker tool* to be used.

In addition, the user has the possibility to fine-tune the verification workflow by setting some parameters (e.g. the settings for the mode selection reduction, or the variables treated as inputs). The mandatory parameters are set automatically using various heuristics.

The verification case can be edited on a form that is shown in Figure 3.12. The user should choose a requirement pattern that corresponds to the requirement to be checked. Then the gaps of the pattern have to be filled. Each gap has to be filled by an expression containing constants, variables, and logic, arithmetic, and/or comparison operators.

Verification process. After loading or writing the PLC code and providing the verification case, the verification procedure is fully automated. In the background PLCverif performs the following steps, completely hidden from the user:

1. The formal requirement (in LTL or CTL) is produced based on the given information.
2. The PLC code is parsed and translated into an intermediate model.
3. The intermediate model is reduced. The reductions depend on the given requirement, consequently, for each requirement a unique verification model is produced.
4. The reduced intermediate model and the given requirement is then converted to the input syntax of the selected model checker tool.
5. When the model and the requirement are produced, the model checker tool is invoked. All its outputs, including the error messages are stored for the later report generation.

Evaluating the results. The output of the model checker tools (the diagnostic traces) are typically difficult to understand. For example, the counterexamples can be extremely large, they have to be reduced before manual analysis. Besides, the referred variable names can be different from the ones

defined in the PLC code due to the automated generation process and the various restrictions of the formalisms. These have to be replaced with names that are meaningful to the user.

The result of this phase is the verification report (see Figure 3.13 for example): a self-contained document produced automatically that includes the details of the verification case, the result of the verification and the reduced diagnostic trace, if applicable.

Continuous verification. Besides the graphical user interface, PLCverif offers a command line interface too. It allows to set up a "continuous verification" (regression verification) workflow implemented using Jenkins[13], where a job automatically rechecks all verification cases on each commit to the version control system, then the results are sent by e-mail[14]. This ensures that every version of the source code is automatically checked, furthermore the verification will not be skipped due to lack of time or computational resources, therefore it helps the acceptance and usage of the verification workflow in practice.

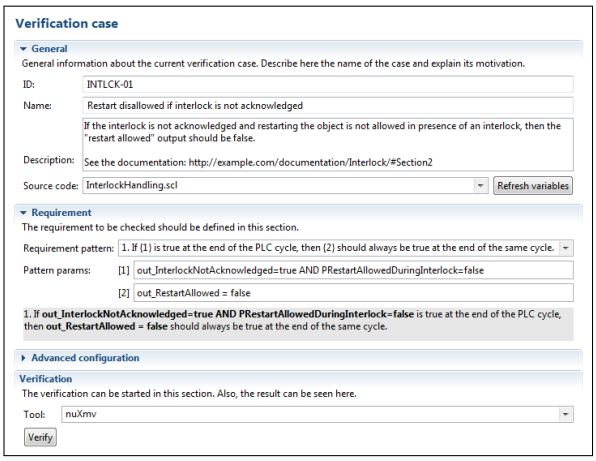


Figure 3.12: PLCverif verification case definition form (based on [c13])

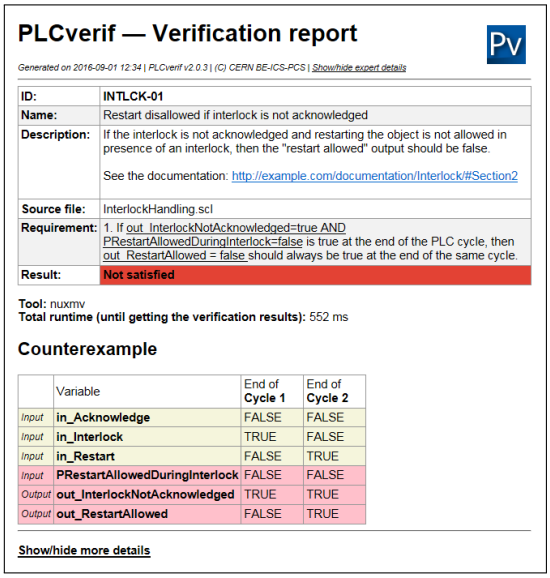


Figure 3.13: PLCverif verification report (based on [c13])

Publications related to this section. This section is based on [c13], a tool paper about PLCverif. Additional discussion about the chosen technologies can be found in the technical report [r24]. A detailed overview of the verification workflow through a real case study is presented in [c16].

3.8 Case Studies

This section is dedicated to the case studies related to the presented verification workflow. First, the verification of non-safety-critical reusable PLC modules is presented in Section 3.8.1, then the verification of a safety logic is discussed in Section 3.8.2.

[13] https://jenkins-ci.org/

[14] The implementation required for the integration of PLCverif with Jenkins was a joint work with M. Lettrich and C. Tsiplaki Spiliopoulou.

The primary goal of these case studies is to demonstrate that the verification of real-life PLC programs is feasible using the proposed verification workflow, furthermore problems can be found in this way. It is also presented that the proposed reduction techniques are useful and beneficial, they make the verification feasible in practice.

Each described measurement was executed on a PC with the following configuration: Intel Core i7-3770 3.4 GHz CPU, 8 GB memory, HGST Travelstar Z7K500 HDD with Windows 7 x64 and .NET 4.0 framework. For the measurements PLCverif 2.0.3 and nuXmv 1.1.1 tool versions were used.

3.8.1 Usage for UNICOS Baseline Objects

One of the original motivations of this work was the verification of the reusable, basic components of the PLC applications used at CERN. This part describes UNICOS, the PLC framework used at CERN and the verification of certain base components of UNICOS.

UNICOS. UNICOS[15] (Unified Industrial Control System) is a CERN in-house framework to develop PLC-based industrial control applications. It mainly covers the supervision and the control layers of the classical industrial process control systems [Bla+11].

This framework consists of a library of generic objects, a development methodology, and a code generation tool. The resource package of UNICOS for continuous process control (UNICOS-CPC[16]) contains 24 generic objects, the so-called *baseline objects*. These basic building blocks represent I/O objects (e.g. digital and analogue inputs and outputs), interface objects (parameters and statuses exchanged between the supervision and control layers), field objects (representations of physical equipments, e.g. valves, motors, heaters, etc.) and control objects (e.g. alarms, PID control). These are the basic components of every UNICOS program. Besides the baseline objects, each UNICOS project consists of application-specific *specification files* and the implementation of *custom logic*. The specification describes the instantiation of the baseline objects and the connections between them. The custom logic contains the application-specific PLC code. Based on the baseline objects and the application-specific data, the source code for the supervision and control layers can be generated automatically. This makes the application development easier and faster, moreover makes the different applications across CERN uniform to ease the maintenance and operation. Furthermore, it reduces the risk of faults by reducing the amount of handwritten source code.

UNICOS-based applications are used widely at CERN in large installations, such as the LHC, particle detectors (e.g. ALICE, CMS, ATLAS) and other experimental facilities (e.g. ISOLDE) mainly for the control of the auxiliary systems such as cooling, gas, cryogenics, and vacuum systems. UNICOS has also been applied in other areas, such as interlock-based applications (e.g. LHC collimators) or motion systems (e.g. winding machines).

As it was described earlier, every UNICOS application is based on the same baseline objects. The source code of these objects has been created manually based on a non-formal specification. As they are common components of every UNICOS application, it is crucial to ensure that the code of these components is correct. Indeed, manual and automated testing is already applied for the baseline objects, however, no formal verification was used since their development.

Verification examples. The verification workflow and PLCverif was used for various parts of UNICOS, and various aspects of these verifications were reported in [*c*16; *j*3; *c*14; *c*15].

[15] Visit http://cern.ch/unicos/ for more details about UNICOS. This introduction of UNICOS is an extended version of an excerpt from [*r*24].

[16] http://cern.ch/ucpc-resources/

Informal description: If there is a rising edge on $\mathcal{A}$, and $\mathcal{B}$ is true, then $\mathcal{C}$ shall be true.

Precise description:

If in Cycle N:	$\mathcal{A}$	is false and $\mathcal{B}$ is true, and
in Cycle $N+1$:	$\mathcal{A}$	is true and $\mathcal{B}$ is true,
then	$\mathcal{C}$	shall be true (in Cycle $N+1$).

Tabular description:

	Cycle N	Cycle $N+1$
Assume (at the end of the cycle)	$\neg\mathcal{A} \wedge \mathcal{B}$	$\mathcal{A} \wedge \mathcal{B}$
Check (at the end of the cycle)	–	$\mathcal{C}$

Figure 3.14: Requirement pattern with rising edge detection

Here an updated version of the verification case discussed in [c16] is presented. This verification case was targeting the *OnOff* UNICOS baseline object that represents a field object with binary state, e.g. a heater, valve or motor. This object can be operated in various modes, and changes between these modes were analysed. A requirement was extracted from the documentation of this baseline object. The informal requirement was formalised using the pattern shown in Figure 3.14.

This allowed the formalisation of the informal requirements. Furthermore, it was possible to automatically generate the IM based on the given SCL program code. Table 3.2 shows the key metrics of the IM before and after reductions. It can be seen that the non-reduced model contains many variables, causing a large potential state space (PSS). The size of PSS is an upper estimation of the reachable state space size, as the state space exploration of the non-reduced model was not possible with the available model checker tools. After the various reductions, the model size dropped significantly, as depicted in Figure 3.15(a) for the key IM elements. Of course, this increases the model generation time, but the difference is negligible compared to the gain in the execution time of the verification. This is a clear evidence of how the reductions can make the verification faster, and in most cases feasible at all.

After the reductions and the generation of the concrete syntax for the nuXmv model checker tool, the given requirement was evaluated in less than a second. The requirement was not satisfied and a counterexample was given. Using the counterexample, together with the UNICOS experts, we identified the cause of this violation. Later, this behaviour was identified as correct and we realised that the documentation of the verified object was incorrect, which was then fixed. A more detailed, step-by-step discussion of this verification case is in [c16], from formalisation of the implementation and the requirement, through model generation and reduction, until finding and fixing the cause of the error.

Table 3.2: Key metrics of the OnOff IMs with the given requirement

Metric	Non-reduced model	Reduced model
Potential state space	3.83×10^{242}	8.72×10^{8}
Number of variables	259	28
Model reduction time	–	0.12 s
Total model generation time	0.09 s	0.21 s
Total run time	*(out of memory)*	0.9 s

The usage of the verification workflow showed real development problems too, additionally to incorrect specifications. In another case the formal verification of a baseline object revealed an issue

which was caused by a missing parenthesis and/or wrong operator precedence assumption in a complex requirement (see Listing 3.2, the outmost parentheses in yellow were not included originally). This problem may block the operation of the represented equipment (e.g. a pump) in certain special cases. When it was first revealed, the experts judged this as a false positive, a condition that is practically impossible. Months later, after a long investigation of a real issue in the cooling system of the LHC, the same root cause was found, proving that the identified problem might occur in reality. After this, the baseline object was corrected and the absence of this issue in the new version was proven using formal verification. The issue occurred once more nearly a year later in production, in a non-LHC experiment's cooling system, where the fix of the baseline object was not deployed yet.

```
1 IF ((PEnRstart AND (E_MEnRstartR OR AuRstart) AND NOT FuStopISt) OR (PEnRstart AND PRstartFS
      AND (E_MEnRstartR OR AuRstart))) AND NOT fullNotAcknowledged THEN
2       EnRstartSt := TRUE;
3 END_IF;
```

Listing 3.2: Extract of the source code causing the violation of the requirement

To facilitate the use of formal verification and to better integrate PLCverif into the development workflow, a continuous verification workflow was set up. Currently, every time a new version of any baseline object is committed to the version control system, all previously defined verification cases are checked on the new implementation. This is completely transparent to the user and it does not consume local computation resources, as the verification is performed on remote servers. If any of the requirements are not satisfied by the new implementation, an e-mail notifies the developers.

These integration and usability efforts together allowed some of our developers to directly use PLCverif. They were able to find, analyse and fix problems in the baseline objects autonomously, without requiring assistance from formal verification experts. The usage of PLCverif and the discussion of the results by the UNICOS baseline object experts raised a wide variety of questions related to requirements, intended behaviour, consistency and correctness of implementation. Thus the advantage of using formal verification is not only to find hidden issues in the implementation, but also an increased understanding of the behaviour, with a special focus on the corner cases.

3.8.2 Usage for Safety Controller[17]

The operation of the LHC at high energies requires strong magnetic fields to bend the particle beams. This is achieved by using superconducting dipole magnets. These magnets should be tested before putting them into production. For this, CERN has a unique testing facility (so-called *SM18 Cryogenic Test Facility*) where the magnets can be tested at low temperature (1.8 kelvins, achieved by liquid helium and nitrogen), high currents (up to 14 kiloamperes) and vacuum. Testing the magnets is a safety-critical task, as a failure may cause serious damage or injury. Therefore a safety instrumented system is in use to allow or forbid the magnet tests based on whether their preconditions are met. The SM18-PLCSE (*PLC pour la sécurité*, safety PLC) is responsible to ensure the safety of some of the magnet tests by allowing or forbidding them based on the current status of the various auxiliary systems (vacuum, cryogenics, etc.). In this project we have applied formal methods from the beginning of the development. The STL implementation (generated from the LAD implementation by the Siemens development environment) of this safety logic is relatively complex, consisting of about 9,500 STL statements. The SCLr representation of this program contains about 125,000 SCL statements, due to the complex logic, resulting in many nesting stack operations.

[17] The following introduction and the measurements are based on Sections 1 and 4 of [*c*9].

After the successful representation of the safety logic in SCLr language, we have captured pattern-based requirements from the tabular specification provided by the client of the project. As this was the first safety-critical PLC program verification project at CERN, the requirements were extracted by the author, closely collaborating with the developers, rather than by the PLC program developers autonomously. In total 24 different requirements were extracted and formalised using requirement patterns. Some of them are fairly simple, while some others contain references to up to 50 different variables. In these cases even with the help of the requirement patterns, it was difficult to express the intended requirements.

In each case the verification was successfully executed, thanks to the requirement-specific and general reductions that reduced both the number of variables and automaton locations. These reductions were able to eliminate all register-representing variables in every case. The typical verification run time of each requirement was 150–170 s, including the model generation, the model reductions and the execution of the external model checker (nuXmv). In case of some requirements, only a small part of the model was enough for the verification of the given requirement, therefore the reductions were able to eliminate a large part of the IM, resulting a total run time of 4–5 s. The total run time of all 24 verification cases together was 43 minutes. The peak memory consumption of PLCverif was 2926 MB, however as the implementation is in Java, this number is an upper estimation of the required memory. The peak memory consumption of nuXmv was 570 MB. In these cases the reductions performed the significant part of the verification, the external model checker was easily able to cope with the reduced model. Even the longest nuXmv execution time was shorter than 30 s, and in many cases it was less than a second. However, according to our experiments, the model checking could not be possible at all without these reductions.

Some key metrics of the verification of two selected requirements are presented in Table 3.3. Furthermore, Figures 3.15(b) and 3.15(c) show the effects of iterative IM reduction loops. For example, the number of IM locations was reduced from the initial 125,000 to 10 in case of Requirement 1, to 26 in case of Requirement 1.

Requirement 1 is a simpler safety requirement that can be evaluated based on only a fraction of a model, and most of the variables can be easily removed by the COI reduction. Requirement 2 is a complex safety requirement expressing that the main power converter can only be turned on if all the necessary conditions are satisfied.

Table 3.3: Key metrics of the SM18-PLCSE IMs with the given requirements

Metric	**Non-reduced model**	**Reduced model**	
		Requirement 1	Requirement 2
Potential state space	3.4×10^{978}	1.44×10^{18}	1.16×10^{104}
Number of variables	588	10	91
Model reduction time	—	7.6 s	162 s
Total model generation time	12.2 s	9.7 s	167 s
Total run time	*(nuXmv crashed)*	9.0 s	186 s

Similarly to the previous example, a continuous verification workflow was set up here as well to ensure that each new version is verified, also to discharge the development machines from the heavy load of the verification process.

After performing the case study we have concluded that the verification was successful, as it was possible to model and verify the critical part of the PLC program. We have applied an iterative workflow: every time the model checking pointed out a problem, we have suspended the verification

process until the root cause of the problem was identified and fixed. Then the verification process restarted with the new code version. In total 12 issues were directly identified using this verification workflow. We have classified the problems found into the following main categories:

- 4 *requirement misunderstanding* problems. In these cases the formalisation of the requirements pointed out ambiguous or contradictory elements in the non-formal specification provided by the customer, overlooked during implementation.
- 3 *functionality* problems. In these cases the problem could have caused unexpected behaviours, but not dangerous situations.
- 3 *safety* problems. In these cases the problem could have caused dangerous situations, i.e. a magnet test might be permitted when it should not be allowed.
- 2 *mixed functionality-safety* problems.

All these problems were found before on-site testing of the PLC program. As the (re)deployment and the on-site testing are time-consuming operations, model checking provided an efficient verification method. Furthermore, model checking does not involve the use of real hardware, therefore no dangerous situations can happen, contrarily to on-site testing.

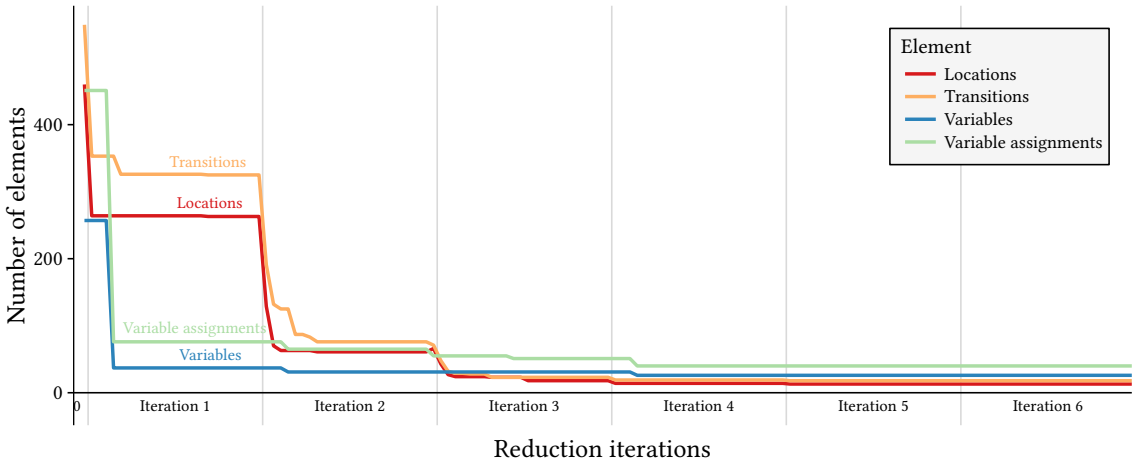
As testing in lab and on site provides the state-of-the-practice in the verification of PLC-based systems, we have checked whether the problems identified using formal methods could have been possibly found using the typically applied testing methods. Setting up a test scenario on-site may take up to hours, therefore only the main functionalities and their most critical errors are targeted, potentially omitting problems. Out of the 8 functionality or safety issues, 4 could have been found using testing. In the other 4 cases it was practically impossible to find the problem using our regular testing approach, as the testing is not exhaustive in practice.

Publications related to this section. The PLCverif model checking workflow was first applied to UNICOS baseline objects in [*r*24]. A detailed verification case study of the UNICOS OnOff object is in [*c*16]. The case study of the verification of the SM18-PLCSE system is described in [*c*9].

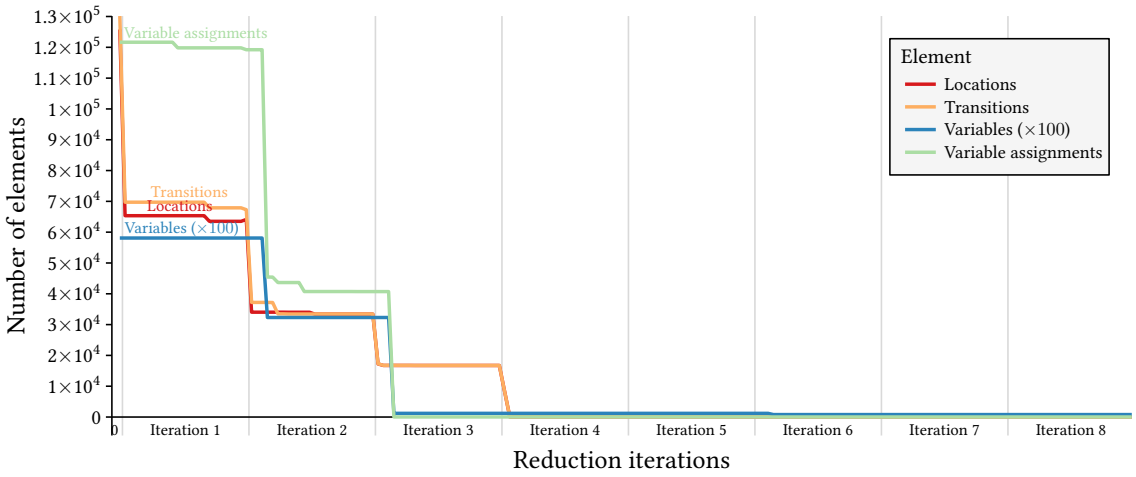
3.9 Related Work

The formal verification, and more specifically the model checking of PLC programs is an interesting topic both in academia and industry. A plethora of various approaches were proposed since the mid-1990s for formal verification of PLC programs. Over the following 20 years various solutions were proposed to verify PLC programs in different languages, using various approaches to ensure scalability. However, the application of model checking is still far from wide-spread in the PLC-based industrial control software development. Many of the proposed solutions are summarised and classified in [Ova+16; FL00]. These surveys focus on the technical aspects of the methods (Which model checker is used? Which PLC languages are supported?). While these are valid and important aspects, the current discussion is focused more on usability aspects. Thus the main question is the following: *Which approaches can be used in practice without excessive effort required?*

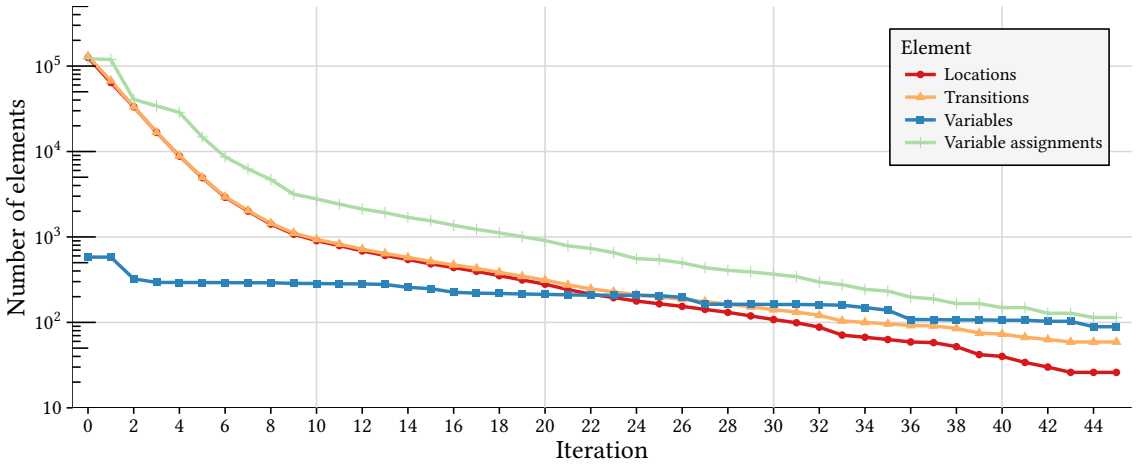
As it was discussed in Section 3.2, for real-life usability the method should not rely on manually created formal models. There are interesting approaches with manual modelling: e.g. Nellen *et al.* [NÁW15] model the SFC program and the model of the controlled plant together with hybrid automata, Soliman *et al.* [SF11] verify the safety function block library defined in PLCopen manually using UPPAAL. However, in the following we restrict the discussion to methods that rely on PLC programs as inputs. The most relevant related works are summarised in Table 3.4.



(a) Effects of the reductions on the OnOff baseline object with the given requirement



(b) Effects of the reductions on the SM18-PLCSE safety logic with the requirement (1)



(c) Effects of the reductions on the SM18-PLCSE safety logic with the requirement (2)

Figure 3.15: Effects of reductions

Table 3.4: Automated formal verification approaches based on PLC programs

Reference	Tool	Supported languages	Demonstrated scalability	Verifier
Barbosa (U. Rio Grande do Norte, BR) [Bar12]	–	IL, ST, SFC, LD, FBD	•	ProB
Biallas *et al.* (RWTH Aachen, DE) [BBK12; BKS12; Bia16]	+	IL, ST, STL	•••	Arcade
Canet *et al.* (ENS Cachan, FR) [Can+00]	–	IL	•	CaSMV
Gourcuff *et al.* (ENS Cachan, FR) [GSF08]	–	ST	••	NuSMV
Helder (TU Eindhoven, NL) [Hel16]	–	SCL	••	CBMC, …
Huuck (U. Kiel, DE) [Huu03]	–	IL, SFC	••	CaSMV
Jee, Yoo *et al.* (Korea AIST, KR) [YCJ08; Jee+10]	+	FBD	•••	CaSMV
Lange *et al.* (RWTH Aachen, DE) [LNN13]	–	IL	••	Z3
Pavlović *et al.* (Siemens, DE) [PPK07; PE10]	–	STL, FBD	••	NuSMV
Sadolewski (TU Rzeszow, PL) [Sad11]	–	ST	•	Why
Sarmento *et al.* (U. São Paulo, BR) [Sar+08]	–	LD	•	UPPAAL
Sülflow *et al.* (U. Bremen, DE) [SD08]	–	IL	••	MiniSat
Darvas, Fernández *et al.* (TU Budapest, HU & CERN, CH) [*j*3]	+	SCL, STL, …	•••	*multiple*

Usability and tool support. The main feature of our presented work and the PLCverif tool is that the complete verification workflow is hidden from the user, as this is a key feature for the usability of the approaches. The column "Tool" in Table 3.4 denotes by "+" the methods including tool support *for the complete verification workflow*. In many cases, only the model generation is supported by tools. For example, according to [Huu03], the CaSMV model representation of the PLC programs is generated by their command line tool SFChecker, however it seems that this tool does not provide support for the complete verification workflow.

While the generation of the formal model is one of the most cumbersome phases in the verification, the users should not be exposed directly to the verification tools. Besides PLCverif, only two tools provide such overall support: Arcade.PLC [BBK12] and FBD Verifier [Jee+10].

The closest to our work is Arcade.PLC by Biallas *et al.* [BBK12]. This is a model checker tool supporting multiple PLC languages (different variants of IL and ST) and includes built-in reductions (CEGAR, predicate abstraction). The verification is performed by their verification engine and not by external model checkers, which causes significant differences in the design decisions. "Arcade.PLC is the first tool to combine fully automatic verification, efficient abstraction techniques, support for different PLC programming languages and a graphical user interface" [BBK12]. The scalability and the efficiency of the reduction methods were proven in [BKS12]. The recent thesis of Biallas [Bia16] shed light to many underlying techniques of the Arcade.PLC tool. This work, similarly to the current work, presents an intermediate representation and various model reductions. According to [Bia16], besides CTL formulae, Arcade.PLC supports the evaluation of requirements given as *safety automata*, aiming to facilitate the requirement specification for non-expert users.

The FBD Verifier tool provides also support for various phases of the verification workflow: model generation (including the FBD to Verilog translation) and counterexample visualisation are discussed in [Jee+10].

Neither Arcade.PLC, nor FBD Verifier provide adequate solution for the user-friendly requirement specification that could hide temporal logic expressions from the users. The lack of support for SCL is another aspect why these tools could not be used in the practice of CERN. By using the requirement patterns, our method implemented in PLCverif covers the complete verification workflow hiding all formal details.

Languages. Most of the methods focus on single PLC languages, as it can be seen in column "Supported languages" of Table 3.4. Exceptions are [Bar12] (supporting all standard languages, but assuming that they are provided in a convenient XML representation), [BBK12] (supporting IL, ST and SCL) and [PPK07; PE10] (supporting STL and FBD via STL). Furthermore, most related approaches support the standard version of the languages [I61131-3], only [BBK12; Hel16; PPK07; PE10] consider the differences between the standard language definitions and the Siemens implementations. While this is understandable from the scientific point of view, the lack of support for e.g. Siemens PLCs and for the variants of the ST language reduce the wider applicability of any model checking method. Both SCL and STL contain features compared to their standard equivalents (ST, IL) that need special support.

Our presented verification workflow provides support for multiple languages thanks to its intermediate model, also to the native support of SCL which is not present in any other known approaches besides the recent work [Hel16].

Intermediate representations. The approach presented in this thesis is not the only one using an intermediate verification model. Arcade.PLC contains an intermediate representation (IR; cf. [Bia16, Sec. 3.4.4]). However, this targets abstractions and representations for SAT solvers, while our IM aims to be suitable for external model checker tools. Šusta [Šus03] formally defined APLC, an abstract PLC machine, which can serve as an intermediate representation between the real PLC code and a timed automaton representation. Syntactically and semantically APLC is close to low-level IL (STL) language. This approach is not a complete PLC model checking solution, thus it is omitted from Table 3.4.

Reductions and scalability. To compare the related works, a key aspect is the scalability of the approaches. Many of the described verification workflows do not provide any specific way to reduce or optimise the models, to simplify the verification problem. Typically, these methods cannot scale up to real, industrial cases.

Certain works paid special attention to the reductions and the scalability. In [GSF08] a compact modelling was suggested for PLC programs, where practically the output variable values are directly expressed using the input values and stored values. This was proven to be an efficient modelling for some cases, but it also imposes certain restrictions (e.g. loops and non-Boolean variables are not supported). Additionally, blocks with complex logic (such as the UNICOS baseline objects) may result in highly complex expressions using this method. [BBK12; Bia16] provide counterexample-guided abstraction refinement (CEGAR), predicate abstraction and slicing. As they rely on their own model checker, the reductions are more tightly integrated with the verification engine than in our case. The solution in [LNN13] implements constant folding based on interval analysis, program slicing, forward expression propagation, etc. Some of these reductions are similar to the ones included in the current work.

The implementation of the referred works are not available (except Arcade.PLC), therefore the assessment of the scalability can only be done based on the case studies and usage examples described in the cited papers. For this reason, instead of a precise comparison, only a three-level scale is used in the "Demonstrated scalability" column of Table 3.4 (where • means the lowest level of demonstrated scalability, ••• the highest). For example, [Sad11] provided an example with 15 lines of ST code, [LNN13] with up to 250 IL statements and [YCJ08] with thousands of function blocks and variables.

Thanks to the reduction heuristics included in our verification workflow, it was possible to verify for example the safety logic of the SM18-PLCSE system, whose implementation consists of about 9,500 STL statements that were translated into more than 120,000 SCLr statements (see Section 3.8.2).

Verification approach. Many of the works in Table 3.4 parse the PLC programs, which are then translated to state machine-like formalisms, e.g. to NuSMV or UPPAAL models. Some other works translate the programs to ANSI C and verify using existing theorem proving tools ([Sad11]) or verification tools for C programs ([Hel16]). Lange *et al.* [LNN13] perform bounded model checking using the Microsoft Z3 solver. Barbosa [Bar12] constructs B models of the PLC programs and uses the ProB model checker.

The used verification approach influences the requirement specification possibilities too. The approaches using NuSMV, CaSMV or UPPAAL as underlying verifiers (see the "Verifier" column of Table 3.4) can benefit from the CTL and LTL model checking included in the tools (restricted CTL only in case of UPPAAL). Some other methods ([LNN13; Sad11; Hel16]) support only assertions (practically reachability properties only). An exception is [SD08]: they use the SystemC models generated from the PLC code to perform equivalence checking between the implementation and a reference model. The definition of the reference model is not discussed in [SD08].

It has to be noted that according to the reviewed papers, none of these approaches support requirement patterns for the specification of requirements that was found to be helpful in our experience.

Semantics of STL programs. Certain works targeted the verification of STL (Siemens IL) programs. For example, Sülflow and Drechsler [SD08] discussed the problem of STL semantics when they translated the STL programs into SystemC for verification. A SystemC-based semantics representation is given for certain instructions, but for example the nested logic statements were not targeted. Pavlović *et al.* [PPK07] target the formal and informal STL semantics too and provide similar representations as presented here in Appendix D.2 (p. 137). However, determining the precise semantics despite the lack of formal semantics definition is not discussed, and therefore the nesting stack was not targeted in their work. Meulen [Meu10] provides formal semantics for the STL language. The discussion of the instructions is more complete here than in [PPK07; SD08], but the instructions with more complex semantics are not targeted here either.

Contrarily, the verification workflow presented in this dissertation provides a simple way to systematically explore the semantics of the STL instructions. By doing this for the Boolean and arithmetic instructions, the verification of certain safety-critical Siemens PLC programs was made possible.

3.10 Summary and Future Work

This chapter discussed a verification workflow specifically targeting the model checking of PLC programs. The workflow is centred around an intermediate model that makes the method easily extensible. It also helps to apply reduction techniques, improving the performance of the model checking.

This verification workflow was extended to support safety-critical PLC programs through the STL language. This required the identification of the precise semantics of STL and development of new reductions.

The implementation of the verification workflow and the real-life usability were demonstrated by showing real use cases where PLCverif, the implementation of the verification workflow was able to identify problems in the requirements and the implementations.

Correctness of the approach and implementation. In this chapter the proof of correctness was not targeted. It is a difficult task to formally prove the complete workflow and its implementation. The translation between the PLC program code and the intermediate model, the model reductions, the translation from the IM to the concrete syntax of the model checkers and the external model

checkers were tested on numerous real examples and special examples implementing corner cases. More thorough verification would be possible for the parts of the workflow which are implemented by me, but not for the external model checker tools. However, as the PLCverif tool does not aim to certify or prove the correctness of the checked PLC programs, furthermore the complete workflow could not be proven to be correct (due to the dependence on external model checkers), more extensive correctness checking was not a priority so far. It is worth noting that we have not observed any false positive verification results in real use cases caused by problems with the workflow or the implementation.

The contributions targeted in this chapter were the following.

Thesis 2 I contributed to the development of a generic, flexible workflow to apply model checking to PLC programs without requiring extensive formal methods knowledge from the users. I designed essential parts of this workflow, as follows.

- 2.1 I designed an intermediate model (IM) language that can represent PLC programs and can act as a pivot language for different model checkers. The IM-based model checking is a fully automated method that can be used by developers who are not familiar with formal verification techniques.
- 2.2 I developed heuristics to automatically reduce the size of the intermediate models, making the model checking workflow less resource-demanding.
- 2.3 I extended this model checking workflow (originally developed only for SCL programming language) to support the PLC programming languages used in the development of safety PLCs: FBD and LAD, via the STL language.
- 2.4 I implemented the proposed model checking workflow in the *PLCverif* tool, providing push-button verification to the developers based on the source code of the PLC program and the pattern-based requirement specification. I evaluated the real-life applicability of this workflow using various PLC modules and applications developed and used at CERN.

Thesis 2.1 was discussed in Section 3.3 which presented the IM language used in the verification workflow. The various IM reduction heuristics were shown in Sections 3.5 and 3.6.3 (Thesis 2.2). Thesis 2.3, the extensions to support safety-critical PLC systems (and therefore the STL programming language) was described in Section 3.6. Finally, Section 3.7 described PLCverif, the tool implementing the described verification workflow and Section 3.8 presented real-life case studies about the applicability of the verification workflow (Thesis 2.4).

Future work. Currently, PLCverif is in a prototype phase, therefore it is used only at CERN. Future work includes additional development in order to make it stable, and publicly available. This would open the door to wider usage. The wider usage will necessarily impose PLCverif to additional types of PLC programs, and certainly some of them will require the design and development of new or improved reduction methods and the integration of additional model checker tools, which are also planned for the future. In addition, it is planned to relax the assumptions on the input PLC programs, e.g. to allow the usage of pointers.

The planned generalisation of B-I-Sat will allow to include the work described in Chapter 2 in PLCverif and the detailed performance evaluation of B-I-Sat for model checking PLC programs.

Chapter 4

Formal Specification for PLC Modules

Motivation. Software faults are inevitable part of the any development process. Formal verification, e.g. model checking can be a solution to improve the quality of the produced software. The work presented in the previous chapters demonstrates that model checking can be applied for verification of industrial control software.

However, any verification technique depends on requirements to be verified. "Until we know the right properties, the best verification technology imaginable will do us no good."[1] This is an important challenge in the verification of PLC programs too. Most often the available specification is high-level, informal and ambiguous; sometimes non-existing or out-of-date. Incomplete, ambiguous and wrong specifications are the sources of many bugs in every software. Certain authors estimate that up to (or over) 60% of the software faults are made in requirements and design phases [SW89].

In Chapter 3 the usage of requirement patterns reduced the gap between the available informal requirement descriptions and the formal requirements needed by the model checker tools. However, this does not solve the issues related to the source of the requirements. In many of our previous verification case studies on real systems, the source of the problems was the specification, not the implementation. In other words, we have found many discrepancies between the specification/documentation and the implementation, but because the specification was ambiguous, partial or incorrect. This finding does not mean that the implementation is correct and only the specification is wrong, that – from the operation point of view – would not be a serious issue. It rather shows that the precise specification is a bottleneck of formal verification, and it is difficult to extract or formalise complex requirements against which the implementation could be checked.

The lack of precise specifications and therefore the missing detailed verification of the implementation is especially crucial in two cases: if a program module has a higher criticality (e.g. safety-critical PLC programs), or in case of often-reused modules (e.g. UNICOS-CPC baseline objects), where a fault may affect a large number of installations. Therefore in this chapter we are focusing on the specification of PLC modules and simple safety logics, rather than on large, complex PLC applications.

Use cases. Various use cases of formal specification are targeted in this chapter. The most obvious of them is the development of new software, where in the future formal specification could be used instead of the current practice of using informal, textual descriptions. Besides, the methods proposed should provide solutions for already existing (legacy) applications, where the implementation cannot

[1] Quote from Pamela Zave. Personal communication, 9 August 2014, Marktoberdorf.

be changed. Furthermore, situations should be kept in mind in which the manual implementation is mandatory, e.g. for safety-critical PLC programs.

Goal. The goal of the work presented in this chapter is to propose a behaviour specification language that is adequate for the specification of PLC modules. These specifications should be formal, but easy to understand, specific for the domain; improving the quality of the specified software and the communication between the customers and the developers (process and control engineers) at the same time.

Such specification language should target various use cases and provide different advantages:

- An appropriate formal specification language can help the *precise description* of requirements which can increase the understanding of the behaviour, and therefore aid the development and the communication with the clients.
- The *well-formedness and consistency* of a formal specification can be checked, furthermore the satisfaction of pre-defined *invariant properties* may be precisely analysed.
- A complete, formal specification can be the source of *code generation*, reducing the need for manual work.
- In some cases code generation cannot be applied. To check the correspondence between the implementation and the specification, *conformance checking* can be used.

It is worth noting that by complete specification we do not mean specifying everything, especially because "[i]n general, it is impossible, when writing specifications, to include everything you want" [Smi85]. We are only focusing on describing the complete behaviour of PLC programs, i.e. the expected outputs for any inputs in each state.

A plethora of formal specification languages has been proposed over the last decades. However, the methods to be proposed in this chapter follow the same principles as in Chapter 3: they should be practice-oriented, easy-to-use, and adapted to the targeted domain [*c*8].

Structure of this chapter. The structure of this chapter is as follows. Section 4.1 overviews the general and domain-specific requirements for a formal specification language that can be used for PLC modules. Section 4.2 is dedicated to the related work on formal specification and conformance checking methods, with a special focus on the PLC-related languages. Section 4.3 introduces PLCspecif, a novel formal behaviour specification language for PLC modules. This section discusses the syntax and semantics of the language. Section 4.4 presents some of the verification possibilities provided with PLCspecif. Section 4.5 describes the code generation based on a PLCspecif specification. Section 4.6 introduces conformance relations and the method for conformance checking. Section 4.7 provides evaluation and real-life examples of the usage of PLCspecif. Section 4.8 concludes and summarises the chapter.

4.1 Requirements Towards a Specification Language[2]

This section overviews the general and domain-specific requirements which have to be satisfied by a formal specification language in order to be practically usable in the PLC software development. The already existing specific formalisms will be discussed later, in Section 4.2.

[2]This section is a extended and modified version of the Section II of [*e*19].

4.1.1 General Requirements

Previous work in the literature can point to necessary requirements and best practices of developing a new, domain-specific specification language. In this part the most relevant general works are summarised.

First of all, a (formal) specification should satisfy obvious general requirements, e.g. it has to be correct, unambiguous, consistent and verifiable [I830].

A formal specification by itself may help the development, however the support for verification can largely improve the quality of the specification and the implementation. For example, support for consistency and well-formedness checking may reveal specification problems and could ensure that the specification is not contradictory. Supporting the definition and verification of invariant properties can further improve the quality of the specification. Equivalence and conformance checking between the implementation and specification can prove that the implementation matches the specified behaviour.

In 2000, van Lamsweerde published a survey [Lam00] on existing formal specification languages and a roadmap for the future. The conclusion of the paper is that "formal specification techniques suffer a number of weaknesses". Such weaknesses are for example:

- Limited scope (i.e. the specification can only capture a part of the system),
- Poor separation of concerns (i.e. the intended properties, the environmental assumptions and the properties of the application domain overlap),
- Too low-level ontologies,
- High cost, and
- Poor tool support.

Van Lamsweerde states that future formal specification languages should be *lightweight* (i.e. not requiring deep formal methods expertise), at least partially *domain-specific*, structured, multiparadigm and multiformat (i.e. integrating multiple languages and letting the specifier use the best for the current needs, thus for each subsystem the most appropriate language shall be chosen, or different languages might be necessary for specifying functional and extra-functional requirements). The general advices like the "average software engineer" should be the target (not the formal methods expert) or that the specification should "provide reasonably fast and visible reward" seem to be obvious, but often overseen. The expected benefits of formalising the specification method are "a higher degree of precision in the formulation [...], precise rules for their interpretation and much more sophisticated forms of validation and verification" [Lam09]. Van Lamsweerde writes also that "there is a long way to go before formal specifications can be used by the average software engineer to provide reasonably fast and visible reward." Despite this statement being made 15 years ago, it seems to still be true.

Knight *et al.* approached the question "Why formal methods are not used widely?" more practically. In [Kni+97] they designed an evaluation framework to assess formal specification languages. Furthermore, they selected three specification languages (Z, PVS, Statecharts) and applied them for a subsystem of a nuclear reactor. Then, the specifications were assessed by nuclear engineers and by developers not expert in formal methods. After a short training period, the general idea and the main advantages of formal specification were welcomed and understood. For the nuclear engineers Z and PVS were too complex, but Statecharts were claimed to be effective for communication and easy to learn, although difficult to search and navigate. The authors emphasise that often overseen features are also recommended to help the industrial usage. These comprise the support for documentation and readability, e.g. by including free-text annotations connected to the elements of the specification. A possible reason why Statecharts [Har87] are often welcomed by the non-computer engineers can be the fact that it was not developed in a purely academic environment, but in strong collaboration with

avionics engineers, taking their habits and knowledge into account [Har07]. This language was based on the informal, explanatory figures drawn by the engineers, thus the starting point of the language was close to the engineer's way of thinking.

A good example of Statechart-based languages is the RSML (Requirements State Machine Language) formalism. In [HLR98] the authors discuss some lessons learned, like simplicity and readability are "extremely important" [HLR98]. The lessons learned in this project are summarised in [LHR99], where five key problems are listed: *(i)* the large semantic distance between the constructed models and the reviewer's mental model, *(ii)* the difficulty of focusing on requirements instead of implementation (building black box models), *(iii)* the error-prone features of formal specification languages, *(iv)* the expensiveness of producing formal specifications, and *(v)* the inability to detect incompleteness and other problems in the requirements. Our current work focuses on the specification of low-level modules, thus the requirements and the implementation are close to each other, but the rest of the findings are useful.

The relevant work of Teufl *et al.* collects needs of the industrial practitioners towards a model-based requirements engineering tool for embedded systems [TKM13]. In their survey the highest ranked requirements were the support for various different representations, document generation, expression of non-functional requirements, and for maintaining traceability links; not formal verification or automated code generation.

Pang *et al.* presented a recent survey [Pan+16] on user-friendly specification languages that can be used directly by the industrial practitioners. They conclude that "for requirement formalisation, templates and patterns based on domain ontologies can greatly simplify the composition of FSL [formal specification language] formulae." This was also demonstrated in Chapter 3, where the usage of requirement patterns made model checking accessible to the non-expert developers. Furthermore, Pang *et al.* emphasise the need for "syntaxes and notations close to the domain languages used by industry practitioners" [Pan+16] and the importance of tool support and visual languages.

Summary of general requirements. Based on the previous overview, the following key general requirements can be collected.

GRa. **Lightweight method.** The specification formalism should be easy to learn and easy to use without the need of deep formal methods expertise.

GRb. **Domain-specific method.** The specification should be close to the mental model of the users and the practices of the domain. This reduces the required specification effort and improves the usability.

GRc. **Reduced expressivity.** Rich languages can provide rich features, but also more space for problems and they can require longer training period. The reduced expressivity helps to avoid the usage of error-prone features. We are looking for a specification method with a "Goldilocks expressivity": a formalism that is expressive enough for the specification of the targeted systems, but not too expressive, in order to reduce the chance of errors.

GRd. **Variety of supported languages.** This is a consequence of the previous two needs. To provide domain-specific languages with reduced expressivity, multiple formalisms might be needed, as the same description method cannot fit all uses. This way for each specified part the most appropriate formalism can be used. This reduces the need for rich, general-purpose languages.

GRe. **Support for verification.** Support for checking the requirements (consistency, completeness, etc.) is needed.

GRf. **Support for documentation.** Support for documentation and readability is required.

4.1.2 Domain-Specific Requirements

The motivation of this work is a real insufficiency in the PLC domain, therefore it is indispensable to address the existing problems and particularities of the domain. Many specification language-related requirements can be extracted from the previously developed PLC programs and by discussing the challenges with the developers. The main domain-specific needs and challenges are summarised below based on the experience gained at CERN.

DRa. **Events with proper semantics.** Although the execution of PLC programs is cyclic and not event-triggered (interrupt-driven), the concept of events still exists (in a latent way). In fact, many Boolean inputs represent events or external actions aiming to modify the internal state of the controller. Events should be treated as "first-class citizens" in the specification. It is also important to adopt an event semantics that is appropriate for the PLC domain. Due to the cyclic behaviour, we cannot have the assumption which is usual in event-triggered systems that all previous events are fully handled before a new event is triggered. As in a PLC multiple events can happen simultaneously, the priority of events has to be defined. If multiple *contradictory* events happen (e.g. both the "open valve" and "close valve" events occur), the one with the highest priority should suppress the events with lower priorities, but several *independent* events may trigger in the same cycle.

DRb. **Clean core logic.** PLC programs work directly with physical input and output (I/O) signals, therefore a significant part of the programs has to perform *input and output handling*. While this task is unavoidable, decoupling the I/O-handling helps to focus on the core logic. As the PLCs have limited resources, the developers try to minimise the number of variables, but this approach is not needed to be followed in the specification. In the specification, it is important to use "concepts" rather than expressions, i.e. it is better to *define named expressions* (practically internal variables for specification purposes only) defined by expressions on the input variables, and to use these named expressions in the definition of the logic.

DRc. **Hierarchical, modular structure.** To support reuse and the abstract design of specification, the language should provide a hierarchical, modular structure. Hierarchy and modularity helps to "divide and conquer", to have a simple logic in the leaf modules (i.e. modules that are not further decomposed), and to avoid the duplicated specification of the same submodules.

DRd. **Time-dependent behaviour.** As PLCs can have time-dependent behaviour, the proposed formalism should support timed models. This behaviour is generally captured by timers in PLCs. Three types of timers are defined in the corresponding standard [I61131-3]: TP (signal pulsing), TON (on delay), TOF (off delay). Each timer type has a different semi-formal semantics described in [I61131-3]. These standardised timers should be part of the language as modules, because they are widely used and well-known by the automation engineers.

DRe. **Relaxed conformance relations.** Verifiability is an important property of any formal specification language. Equivalence and conformance checking are powerful tools to check the correspondence between the specification and the implementation. However, in case of control software sometimes the strict equivalence is not mandatory, as the reaction time of the physical process might be orders of magnitudes slower than the controller's cycle time. Therefore relaxed, more permissive conformance relations might help the developers to focus on the real differences and exclude the acceptable differences between the specification and the implementation.

Publications related to this section. The general and domain-specific requirements towards an appropriate practice-oriented, PLC-specific formal specification language were discussed in [*e*19], also briefly in [*c*12].

4.2 Related Work

This section overviews the previous work related to formal specification methods. First, the widely-known languages are discussed (Section 4.2.1), after the various earlier equivalence and conformance checking methods are targeted (Section 4.2.2).

4.2.1 Formal Specification Languages

The specification of various systems is always crucial, especially in the field of safety- or mission-critical systems. Thus obviously the different specification methods are deeply studied and various languages are proposed.

General-purpose formal specification languages. There are several well-known, general-purpose formal specification languages that are relatively widely used in computer science. The Formal Specification Languages working group of the International Organization for Standardization (ISO) maintains two standardised formal specification notations: the VDM-SL (Vienna Development Method Specification Language) [I13817-1] and the Z Notation [I13568]. There are other widely-known formal specification notations, such as the Abstract Machine Notation of the B-Method, Alloy[3], communicating sequential processes [Hoa85], or PVS (Prototype Verification System) [ORS92] specification language. These methods require deep knowledge from the user. However, they are not purely theoretical, there are many published industrial case studies: for example, the B-Method was used in the development process of automatic subway systems [Beh+99] or for car diagnostics [Pou03].

The Lustre [Hal+91] and Esterel [BG92] formal languages were made known and used by ANSYS (originally: Esterel Technologies) SCADE, a software suite for model-based control software design, targeting highly critical uses, such as avionics, railway industries or defence. SCADE is a "quite rare success story in the domain of formal methods" [Hal05]. Although the languages and the tools were designed targeting control engineers specifically, in the ICS domain the usage of SCADE would need too high effort and resources, not justified for the experienced level of criticality.

Most of these specification languages have a common issue: their usage needs deep expertise in the field of formal methods and mathematics, thus typically they cannot be used in cooperation with the domain experts. This fact restricts their usage to some special cases where the cost of a failure would be extremely high, and where for this reason involving specialists and higher training costs are justifiable. Therefore these methods cannot be used directly in typical PLC software development processes. Furthermore, these languages are not targeting industrial control software specifically. For this reason, even after considerable training effort the specification using these languages is a difficult, complex task, as these languages do not satisfy most of the domain-specific requirements discussed in Section 4.1.2.

Different extended versions of state machines, such as Harel's Statecharts [Har87] or the UML (Unified Modeling Language) state machines [UML11] are more commonly used and accepted, partially because the knowledge level required for their usage is significantly lower. The original Statechart formalism suffered from some serious problems, which induced new formalisms: Beeck in 1994 compared already 20 different variants of Statechart [Bee94]. An interesting specification language based on the Statechart formalism is RSML (Requirements State Machine Language) which extends Statechart among others with tabular expression definition methods. It was used for the specification of the TCAS II (Traffic Collision Avoidance System) of airplanes [HLR98]. Tabular notations, providing cleaner expression definitions, were already proposed by Parnas in [Par92], then later for state

[3] http://alloy.mit.edu/alloy/citations/case-studies.html

machines in [HKB08]. A common weakness of the Statechart variants is the lack of unambiguous formal semantics and that they cannot capture all kinds of requirements efficiently (e.g. data-flows), thus they are not suitable on their own for the specification of arbitrary industrial control software. Moreover, the semantics is not defined formally, although there were various attempts to formalise different subsets of the Statechart or UML state machine formalisms, e.g. [LMM99; Pin07; DDd03; Liu+13; Bee02]. Furthermore, the provided solutions do not fit to the PLC domain and using only a Statechart-like formalism does not provide a convenient method for specifying PLC programs, e.g. behaviour of modules with many numeric state variables are difficult to be captured.

The Simulink Stateflow is another Statechart-like, widely-used formalism. It originally does not have a formal semantics definition, but later attempts were made for its formalisation [Ham05]. Initial experiments showed that it has a semantics even richer than the original Statecharts which – without special adaptation to the industrial controls software – makes its usage error-prone and cumbersome.

Other works target the problem of describing individual requirements or scenarios instead of providing complete behaviour specifications. CTL and LTL were already introduced in Chapter 2 as languages capable of precisely describing requirements. Improvements and extensions to expressivity were proposed for example in [EF18] to make temporal logics more practical. The Property Specification Language (PSL) is a more user-oriented assertion language that can describe desired invariant properties, mainly targeting concurrent systems [FMW05; Gla+07]. Requirements can also be described using various requirement patterns [DAC99; CMS08], as discussed before. Other languages, such as the Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs) provide formalisms to describe visually possible, necessary and forbidden behaviours [HT03; DH01].

These languages provide means to describe individual requirements. This is definitely useful for the formal verification, but our motivation for using formal specification languages is to provide a complete description of the behaviour of certain PLC modules. Therefore these languages are not directly applicable in our case.

After realising that the widely-used specification languages do not provide appropriate solutions for the considered use case, the focus was set to domain-specific formal specification languages, specifically targeting the industrial control software.

PLC-specific specification languages. Even though there are many clear advantages of using formal specifications, this is not a common practice for the development of industrial control software. A survey for the MEDEIA European project in 2009 stated that the most widely used specification tool for these systems is Microsoft Word [CLS09]. Semi-formal methods, such as UML are used in addition in some cases. Writing textual specifications may not need special expertise, but its ambiguity can cause misunderstanding, incorrect implementation, thus increased verification and maintenance effort.

Several researches target specifically the (semi-formal or formal) specification of PLC programs [Lju+10; KLC06; ZGS11; GCG10; Wan+09] (see Table 4.1 for a brief summary). Even if they contain valuable ideas, *(i)* they propose rather mathematical formalisms that need high level of expertise in the formal methods field; *(ii)* the semantics is not clear or not appropriate; or in other cases *(iii)* the solutions are too close to the implementation level.

The column "Easy to learn" is a subjective classification of how easy it is to master the formalism (● denotes the most difficult, ●●● the easiest). The "Precise semantics" column indicates whether there is a precise, formal semantics definition available. This also determines if the formal verification is

possible based on the method. The "Code generation" column indicates whether the method incorporates automated implementation synthesis (+) or not (−). The "*" indicates that restrictions apply, detailed in the following.

Grafcet is a widely-used, standardised [I60848] specification language that has been developed for about 40 years. Even though Grafcet has its roots in Petri nets, it does not have a defined formal semantics and various authors proposed different formal semantics [PRF11a]. Furthermore, it targets only certain specific types of control software, therefore it could not be used in our development process. The SFC language defined in [I61131-3] is based on Grafcet, and has a wide tool support. As SFC programs can be directly executed on most PLCs (or compiled to the appropriated format), Grafcet is implementable in practice. However, the semantics of the different SFC variants do not fully match the semantics of Grafcet.

ST-LTL [Lju+10] is a PLC-specific adaptation of the LTL formalism. This is a formal language that can be easily checked on the implementation, as the requirements are close to the formal requirements needed by model checkers. On the other hand, the ST-LTL formalism is not close to the automation engineers' general knowledge. Furthermore, it is difficult to scale with the growing size and complexity of the module to be specified, and it might be difficult for the developer to see whether the specification is consistent and complete [*e*19].

Another method relying on LTL is *G4LTL-ST* [Che+14]. It uses a set of LTL requirements and based on them it synthesises a satisfying implementation in standard ST language. While the promise of automated code generation based on declarative requirements can be appreciated, defining precise behaviour with a set of LTL requirements is extremely difficult even for users experienced in formal methods. In an experiment small snippets of real PLC code used at CERN were specified using LTL. By the time the specification became complete it was too complex to handle and the generated implementation was also hundreds of times longer than the original code.

NuSCR, based on SCR (Software Cost Reduction) is a formal specification language targeting the requirements of real-time embedded software, especially in nuclear domain [Yoo+05a; Yoo+05b]. The NuSCR method is incorporated in NuSEE (Nuclear Software Engineering Environment), an "integrated environment for software specification and V&V for PLC based safety-critical systems" [Koo+06]. NuSCR provides a visual, state machine-based formalism with tabular extensions, then based on the specification automated generation of PLC program in FBD. In [Yoo+05b] a formal semantics definition is also provided. However, it seems that NuSCR does not provide appropriate solution for (sub)modules that are difficult to describe using state machines. Moreover, the state machines of NuSCR can easily become complex, thus error-prone. Furthermore, the level of details given in the cited paper does not allow a precise evaluation of the method.

ProcGraph [GLK13; Luk+13] is a recent approach to the specification of PLC programs. This is also a state machine-based solution that can be convenient for certain cases, however based on the presented examples it seems that ProcGraph provides only a partial solution for specification and a large amount of PLC code is included directly in the specification. ProcGraph provides means to generate PLC programs implementing the specification automatically, but the lack of formal semantics definition reduces the possibility of verifying the specification.

A similar, state machine-based specification is *GrafTab* [So95], but it lacks the formal semantics definition too. Furthermore, it includes only an informal overview of the implementation, but it seems that the automated implementation generation is not targeted.

It is worth to mention the safety automata formalism used in Arcade.PLC to specify safety requirements. This is a non-deterministic automata-based notation that can recognise all acceptable behaviour of a program [Bia16]. The expressivity of this formalism is reduced, thus it does not aim to

Table 4.1: PLC-specific specification languages

Language/Method	Base formalism	Easy to learn	Precise semantics	Code generation
Grafcet [I60848]	Petri net	●●●	−	+*
ST-LTL [Lju+10; LÅF10; Lju11]	LTL	●	+	−
G4LTL-ST [Che+14]	LTL	●	+	+*
NuSCR [Yoo+05b]	state machine, tables	●●	+	+
ProcGraph [GLK13; Luk+13]	state machine	●●	−	+
GrafTab [So95]	state machine, tables	●●	−	−*
PLCspecif	multiple	●●	+	+

be a complete specification method, instead it facilitates the description of safety properties for model checking.

Many authors provided UML or SysML-based specification, design and implementation solutions, e.g. [VWK05; TF11b; Vog+14; Häs13]. Even if they can help to reduce the semantic difference between the design phase and the implementation phase, an important gap remains between the modelling language and the developer or customer. UML is often considered to be part of the "common knowledge" of computer and software engineers, thus it can be used as a base language. However, this is not true for the targeted domain, as "UML is not familiar to most process control practitioners and is perceived as too complex" [Luk+13]. The UML or SysML-based approaches cannot be fully adapted to the domain or reuse the already used formalisms, thus we do not further discuss these approaches. A more complete overview of the various UML or SysML-based approaches can be found in [Luk+13].

Besides, certain specification works target explicitly the PLC-based systems based on the IEC 61499 standard [I61499-1]. As this standard is not widely supported by the PLC vendors yet, we do not consider these approaches appropriate for our usage.

Others (e.g. in [TF11a]) try to develop specification methods based on the piping and instrumentation diagrams (P&ID) that is an IEC standard diagram [I62424] representing the process equipments and instrumentations along with their interconnections. Although P&IDs do not carry all the necessary information for the implementation of a control software, this might be an interesting and appropriate approach for the user-friendly specification of complete control applications. However, it does not provide solution for the specification of PLC modules, which is the target of this work.

4.2.2 Equivalence and Conformance Checking[4]

Equivalence and conformance relations have a long history. Already in 1981, Back summarised and described various refinement and equivalence relations in [Bac81]. Tretmans described the ioco relation [Tre96] a decade later to check the conformance between a specification and an implementation. [Lam+11; Pha13] are recent works introducing newer relations responding to various needs. However, all of these works target mainly reactive systems, where the level of required conformance is typically high. In cyclic transformational systems, such as PLC-based control systems this leads to numerous discrepancies that are considered to be "false positives", i.e. not relevant from the practical point of view.

Sülflow and Drechsler [SD08] applied strict, non-timed equivalence checking based on a SAT solver to verify PLC programs against their specifications. Provost *et al.* [PRF14] check the strict conformance between a specification (given as a Mealy machine [PRF11b]) and an implementation by

[4]This discussion is based on the Section VI of [*c*6].

generating test sequences. Equivalence checking between FBD programs were targeted through Verilog in [YCJ09], between FBD programs and ANSI C programs in a similar fashion in [LYL11]. Besides these works, applying equivalence and conformance relations specifically to PLC-based systems was not targeted until the very recent works of Weigl, Beckert, Ulewicz *et al.* [Bec+15; Ule+15; Wei15]. Their goal is to perform regression verification on PLC programs, i.e. to check the conformance of two different PLC programs. For this purpose two relations are defined: the *perfect equivalence*, when the two PLC programs should give the same output sequence if the same input sequence was given; and the *conditional equivalence*, which relaxes the requirements of perfect equivalence: the two implementations should produce the same output only if a certain condition is met (practically the two implementations can give different output for impossible input scenarios).

One of the goals of their work is to complement testing by defining and checking sensitive conformance relations. For example, a one-cycle-long delay in the output might mean a delay of 1–100 ms on the output which needs high precision measurements to be checked using testing. However, in many cases such a small delay does not affect the controlled process in an observable way, therefore in these cases the two compared artefacts can be considered as conformant to each other despite the minor differences. Therefore more permissive conformance relations are needed.

4.3 Syntax and Semantics of PLCspecif[5]

Based on the analysis of the requirements and the related work, no suitable formal specification method was found that could help the development of ICS in practice without excessive effort. This observation resulted in a project of designing a new, PLC-specific specification language according to the requirements discussed before. This section describes PLCspecif, the result of this work: a new behaviour specification language for PLC programs.

The discussion starts with the structure of the PLCspecif specifications (Section 4.3.1). Then the expression description methods (Section 4.3.2) and the core logic description methods (Section 4.3.3) are presented. Finally, an informal semantics is presented in Section 4.3.4.

This section provides only an informal overview of the specification language. For the detailed formal syntax and semantics definition the reader is referred to the technical report [*r*22].

4.3.1 Structure of the Specification

The main building block of a PLCspecif specification is the *module* that is either a *composite* module (its behaviour is described by several submodules) or a *leaf* module (its behaviour is directly described). To provide a specification method that is familiar for the PLC developers, the leaf module descriptions are based on three widely-known formalisms: state machines, data-flow diagrams, and standard PLC timers. Typically the behaviour described by the composite modules is the sequential composition of the submodules' behaviours. An exception is the alternative module, where based on a condition one of the submodules will determine the behaviour.

Each module (both composite and leaf modules) is further decomposed into three main parts: *(i)* input definitions, *(ii)* core logic description, and *(iii)* output definitions. According to the semantics of PLCspecif, these parts are executed sequentially in a loop, following a structure similar to the cyclic execution scheme of the PLCs.

In the *input definitions* part, named expressions can be defined to simplify the specification. For example, if there are three digital inputs representing three buttons (`Button1`,

[5]This section is an adapted excerpt from [*c*12].

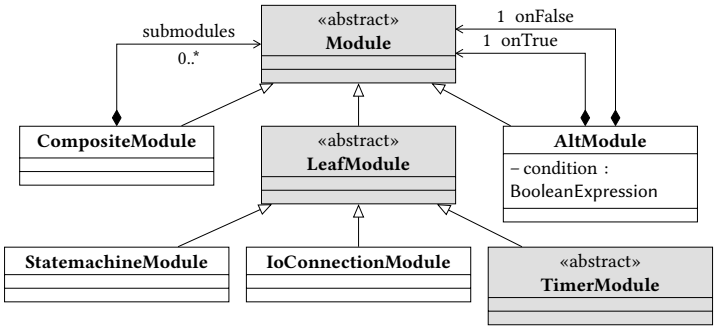


Figure 4.1: Abstract syntax (metamodel) of PLCspecif module structure [*e*19]

`Button2`, `Button3`), but the program provides the same response to pressing any button, writing `Button1 OR Button2 OR Button3` in the core logic makes the understanding more difficult. Instead, the user can specify a `ButtonPressed` expression that is defined once and used later in the core logic. This helps to decouple the physical structure (i.e. three digital inputs) from the concepts to be used in the core logic (i.e. one of the buttons was pressed). Some special inputs, called *event inputs*, can also be defined. An event input is an expression with Boolean type that has a priority assigned.

The input definitions are followed by the *core logic description* part. Several formalisms can be used here that are introduced later, in Section 4.3.3.

The *output definitions* part is responsible to assign values to the output variables, based on the input values and the core logic (e.g. the current state of a state machine). This helps to keep the core logic clean. Including the output variable assignments in a state machine (as entry/exit actions or as transition actions) might make the state machine difficult to overview, therefore it is error-prone[6].

Optionally, the output definitions part can be followed by *invariant properties*. These are additional requirements and assumptions identified during the specification phase that are not obviously described by the core logic, but have to be satisfied by the module.

4.3.2 Expression Descriptions

The input or output definitions may contain complex expressions. While the arithmetic form is suitable to describe simple expressions (e.g. `a OR b`), it does not scale up well. PLCspecif supports the usage of other expression description methods: AND/OR tables and switch-case tables.

AND/OR tables were introduced in the RSML formalism [Lev+91], but were not widely used since. In an AND/OR table each column represents a case, which is true if in all the rows the value of the expression in the row header equals to the value in the corresponding cell of the case. The whole expression is the disjunction (or-connection) of the defined cases. The symbol "·" denotes a "do not care" value. Figure 4.2(a) shows an example, representing the `a AND NOT b AND (c OR NOT d)` ($a \wedge \neg b \wedge (c \vee \neg d)$) expression in a tabular format.

Switch-case tables are based on similar principles as the `CASE` constructs of various programming languages. Figure 4.2(b) shows an example, representing `_Value`, limited by lower limit `PMin` and upper limit `PMax`. The formal semantics of these tabular descriptions can be found in [*r*22].

[6]It has to be noted that the timer and input-output connection modules may assign certain variables, while the state machine modules cannot assign any variables.

	Case 1	Case 2
a	true	true
b	false	false
c	true	·
d	·	false

(a) AND/OR table

	_Value < PMin	_Value > PMax	Result
Case 1	true	·	PMin
Case 2	false	true	PMax
Case 3	false	false	_Value

(b) Switch-case table

Figure 4.2: Tabular expression description examples [*c*12]

4.3.3 Core Logic Descriptions

A single formalism cannot conveniently fit the different types of modules (with state-based, data flow-oriented and time-dependent behaviours). Therefore three different types of core logic descriptions are included in PLCspecif: state machines, input-output connection descriptions and PLC timers. They are discussed in the following paragraphs.

In the following we focus on the state machine module due to space restrictions. The input-output connection and timer modules are introduced informally, and for further information the reader is referred to [*r*22].

State machine module. A *state machine module* is composed of hierarchical *states* and *transitions*. A state can be *single* (that cannot be further decomposed) or *composite* (grouping several basic states together). A single state is either a *basic* state or a *deep history* pseudostate. A basic state represents a real state of the module. A deep history state can store the last active basic state in a composite state, similarly to UML State Machines. A deep history state may never be active and it should not have any outgoing transitions.

A transition can go from any state to a single state[7] (basic or deep history state). It can have a Boolean expression as guard condition: the transition can fire only if the expression is evaluated to true. There are two types of transitions: *event-triggered* (that has an attached trigger event) and *non-event-triggered* (without trigger event).

The structure of the state machine module may look similar to UML State Machines. However, there are some important differences, e.g. the event-triggered transitions and their semantics, no actions can be attached to transitions or to entering or exiting states, or the restrictions on the target of transitions. The precise structure (metamodel) of the state machine modules is shown in Figure 4.3. An example state machine can be seen in the core logic part of Figure 4.5.

Input-output connection module. State machines are suitable for modules that are stateful and the state to be stored can conveniently represented by a handful of states in the specification. However, if the state can only be described by some integers or real numbers (e.g. storing previous measurements or value requests), state machines are inappropriate and we propose the usage of *input-output connection modules*. The idea of the input-output connection module is inspired by Function Block Diagrams (FBDs) [I61131-3] and similar data flow-like formalisms. It graphically defines how the outputs of the module should be assigned based on the current inputs and outputs from the previous cycles.

[7] Transitions going to composite states are not allowed as they could make the semantics of a state machine more difficult to understand.

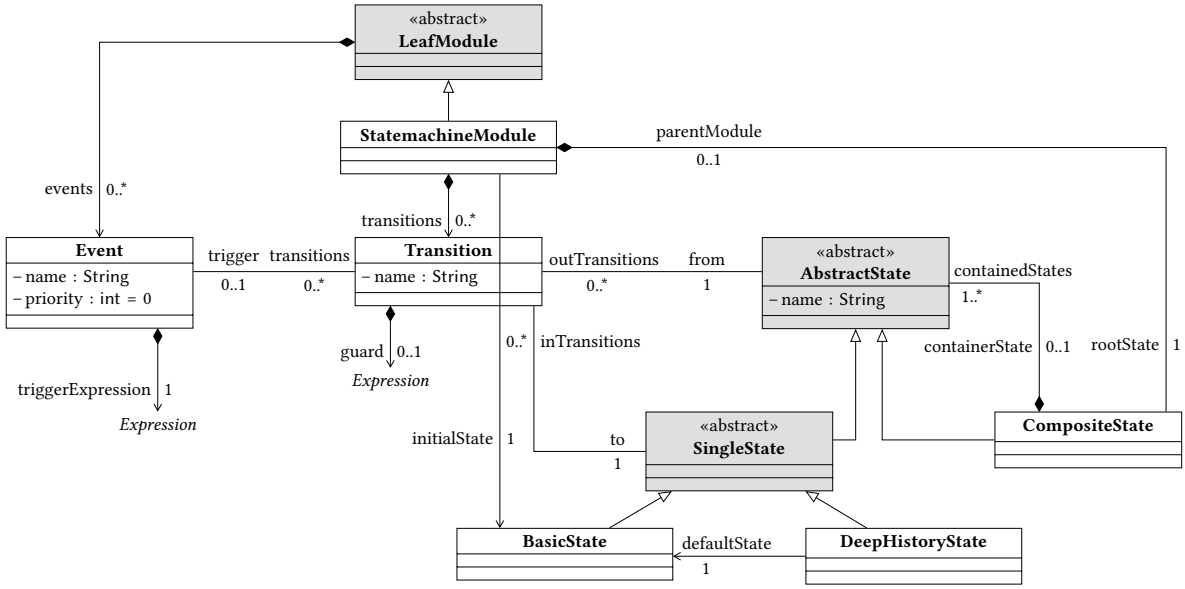


Figure 4.3: Metamodel of the PLCspecif state machine modules (based on [*r*22])

This module description consists of *pins* representing input and output values, and *edges* representing data connections between pins. Furthermore, it contains *blocks*, representing common functions (e.g. logic operations, arithmetic operations, selection), user-defined functions, and platform functions.

Figure 4.4 shows a simple example[8]. Here, the variable `ValueOutput` keeps *its previous value* if the Boolean input `Sample` is false. If `Sample`=*true*, the new value of `ValueOutput` will be $-1 \times$ `ValueRequest`.

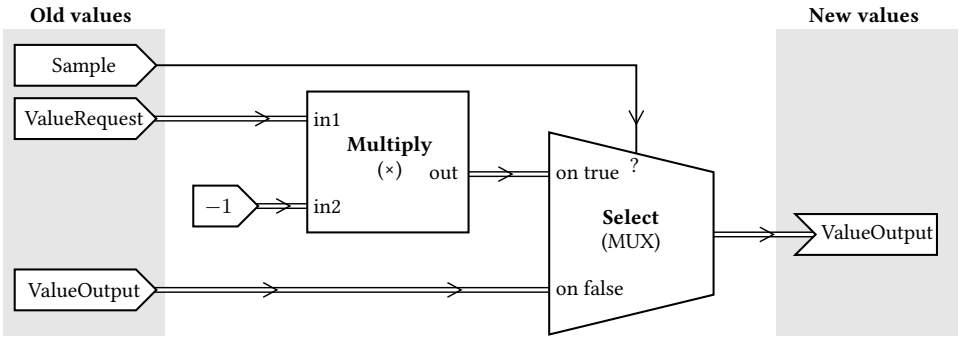


Figure 4.4: Input-output connection core logic example [*c*12]

PLC timers. The state machines and input-output connection modules of PLCspecif do not contain timed behaviours in order to keep their description and semantics simple. However, it is crucial to be able to define time-related operations. State machines are often extended by clock variables to describe time, but this method is error-prone. It also does not fit to the existing knowledge of the target group.

[8]The example is practically an n-bit latch (where n depends on the used data types), which stores the multiplicative inverse of the data input, when it is enabled.

ExampleModule

This module represents value limiter. If it is enabled, the output value is within the limits given as parameters. If disabled, the output is always 0.

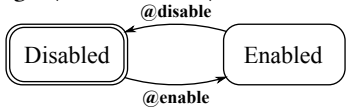
Assigned inputs:
- ValueReq : INT16 *Value to be limited.*
- EnableReq_fromLogic : BOOL
- EnableReq_fromScada : BOOL
- EnableReq_fromField : BOOL
- DisableReq : BOOL
- PMin : INT16 param *Lower limit.*
- PMax : INT16 param *Upper limit.*

Assigned outputs:
- Value : INT16
- Status : BOOL

Input signal definitions: — (none)

Event input definitions:
- @disable ⇐ `rising_edge`(DisableReq) (`pri`=1)
- @enable ⇐ EnableReq_fromLogic OR EnableReq_fromScada OR EnableReq_fromField (`pri`=2)

Core logic (state machine)



Output signal definitions:

- _Value =

`ValueReq` < `PMin`	`ValueReq` > `PMax`	result
T	·	`PMin`
F	T	`PMax`
F	F	`ValueReq`

- Value =

`in_state`(Enabled)	result
T	`_Value`
F	0

- Status = `in_state`(Enabled)

Invariant properties:
- ALWAYS *PMin* ≤ *Value* ≤ *PMax* ASSUMING *PMin* ≤ *PMax*
  If the limit values are valid, the output is always within the limits.

Figure 4.5: Example module specification [*c*12]

Instead, we propose to use PLC timers defined in IEC 61131-3 [I61131-3] (TP, TON, TOF) as separate PLCspecif modules. Their semantics is well-known by the developers and they can use these timers confidently.

Example. *Figure 4.5 shows the specification of a simple state machine module. The described component is a combination of a flip-flop and a multiplexer. If the module is enabled, its* `Value` *output is the* `ValueReq` *input, limited by* `PMin` *and* `PMax`*. The module can be enabled by having a "true" signal in one of the* `EnableReq` *inputs. If there is a rising edge on the* `DisableReq` *input, the module will be disabled, and in this state the* `Value` *output will be 0. Disabling the module has priority over enabling. The module keeps its state if no enable or disable request is received.*

In the example one can observe the structure and general elements of the specification, the decoupled input/output handling, and the different ways of specifying expressions. To help the understanding, each part of the specification can be annotated by textual descriptions.

4.3.4 Semantics of PLCspecif[9]

This part presents the informal semantics of PLCspecif. The detailed formal semantics definition is described in [*r*22]. In addition, Appendix E (p. 145) provides an illustration of the formal semantics by defining the underlying automata formalism, and the pseudocode of the translation algorithm for the state machine modules.

The semantics of the specification language could be formalised in various ways, e.g. by giving operational or axiomatic semantics. Instead a more practical approach was chosen and the formal semantics of PLCspecif is defined based on automata theory, as a construction of equivalent automata. Automata have well-defined formal semantics, and can be easily extended by variables and variable assignments. The semantics of a PLCspecif specification defined as an automaton extended with variables can be mapped easily to the control flow graph (or control flow automaton) of its implementation, helping to design a code generator that follows the formal semantics. Furthermore, by defining the semantics based on automata, this semantics definition is close to the IM language that is used for model checking in the verification workflow presented in Chapter 3. This allows the development of a "PLCspecif to IM" translation which then helps to provide verification methods for PLCspecif, as it will be discussed later in this chapter. The precise definition of the timed automata considered as underlying formalism is in Appendix E.1 (p. 145).

The rest of this section discusses the semantics of the main features of PLCspecif: the general representation of the composite and leaf modules, then the semantics of the state machines and timers. The semantics is described in a top-down manner, starting from the general module representation.

4.3.4.1 General Module Representation

Each module follows the same structure, therefore their semantics (the corresponding automata) are also structured in the same way. The execution of both the leaf and composite modules start with the evaluation of the input signal definitions and (in case of leaf modules) event input definitions. Then, the module-specific part is executed. For a composite module, this is the sequential execution of each submodule in their pre-defined order. For a leaf module, the core logic can be described as a state machine, an input-output connection diagram or a PLC timer description. The semantics of these specific parts are discussed later. After the module-specific part, the output values are set based on the output signal definitions in their predefined order.

These common parts (i.e. the input and output signal definitions) are unconditional variable assignments, therefore they are represented as one single path in the automaton, where the consecutive variable assignments are attached to consecutive transitions. The core logic between the input and output definitions is defined separately for each module type. Figure 4.6 illustrates the structure of the constructed automaton.

4.3.4.2 State Machine Representation

The syntax of the state machine module is similar to any hierarchical state machine formalism, but the semantics is simplified and adapted to the needs observed in the PLC program development. For instance, parallel regions are not allowed in the state machine, implying that there is always exactly one active basic state in the module. The active composite states are implied by the active basic state (all parent composite states of the active basic state are active too), therefore the active state configuration can be represented by the active basic state only. This will be represented in the automaton by

[9] This part is a modified and extended version of Section III-E of [*c*7] and [*r*22].

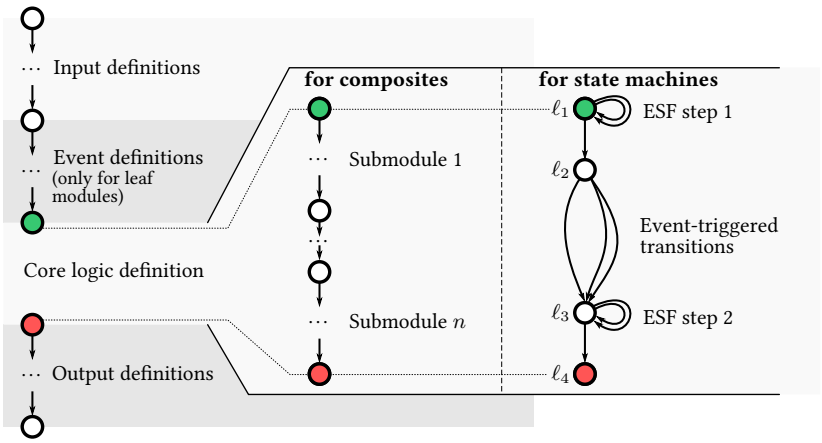


Figure 4.6: Illustration for semantics definition based on automata extended with variables [*c*7]

the variable `activeState` whose type is an enumeration of all *basic states* of the module. The default value of this variable is the initial basic state of the module.

Enablement of transitions. A transition is *enabled*, if *(i)* its source state is currently active, *(ii)* the condition of the transition is evaluated to true, and *(iii)* if it is event-triggered, the connected event triggers (i.e. it is enabled and there is no enabled event with higher priority in that module). When an enabled transition *fires*, the current active state of the state machine is changed to the target state of the transition. A transition firing cannot cause any other effects (e.g. it cannot provoke actions or do variable assignments, unlike for example in UML State Machines [UML11]).

Phases of transition firing. The execution of the state machine (i.e. firing of transitions) has three phases: *(i)* an "exhaustive stabilisation firing step" (ESF step), *(ii)* the firing of at most one event-triggered transition, and *(iii)* a second ESF step. The goal of the ESF steps is to ensure a stable state (to leave transient states and reach a tangible state configuration in which the state machine waits for the next event input), while the event-triggered transitions are the reactions to the incoming requests. Note that in PLC specifications transient states having non-event-triggered transitions are typically used for error handling purposes.

- In Phase 1, the ESF step exhaustively fires all enabled non-event-triggered transitions t_{NT}. Here we assume that the specification respects the well-formedness rules (defined in [*r*22]), including that only a finite number of transitions can fire in any ESF step (thus infinite firing sequences are not possible) and in each state at most one non-event-triggered transition can be enabled (non-determinism is prohibited in the ESF step, but several non-event-triggered transitions may fire consecutively).

  The ESF step is represented by a location ℓ_1 in the automaton (see Figure 4.6) and an $f = \langle \ell_1 \to \ell_1, g, v \rangle$ loop edge[10] for each non-event-triggered transition t_{NT} of the state machine. The guard of t_{NT} is represented by the automaton guard g of f, while the state change caused by t_{NT} (updating `activeState`) is denoted by the assignment v of edge f.

[10] To avoid the confusion, we will consistently use *transition* for transitions that are defined in PLCspecif state machines and *edge* for the state transitions in the constructed automaton.

- If there are no more enabled non-event-triggered transitions, Phase 1 is over. This is represented by an edge $\ell_1 \to \ell_2$ in the automaton. The guard of this edge represents that no more non-event-triggered transitions are enabled. Then, in Phase 2, at most one event-triggered transition fires: the enabled transition that is triggered by the triggering event input. According to the well-formedness rules of PLCspecif, for each state of the state machine module, any event input e cannot have multiple outgoing transitions triggered to e where the guards can be evaluated to true at the same time. For each event-triggered transition t_{ET} we have an edge $\ell_2 \to \ell_3$ with guard representing that *(i)* the source state of t_{ET} is active, *(ii)* the guard of t_{ET} is true and *(iii)* its triggering event input is active. According to the assumption above, this automaton representation does not have any non-determinism. If none of the event-triggered transitions can fire, we proceed to the next phase without any state modification.
- Finally, in Phase 3, a second ESF step finishes the execution of the state machine in ℓ_3. This step can be skipped if there was no event-triggered transition firing. The $\ell_3 \to \ell_4$ edge with its guard represents that the second ESF step is over, there are no more enabled non-event-triggered transitions.

The principles discussed up to this point are illustrated in Figure 4.6. More information can be found in Appendix E.2 (p. 146) about the construction of a corresponding automaton for a PLCspecif specification.

4.3.4.3 PLC Timers

We recall that PLC programs defined in PLCspecif may have timed behaviours that are isolated and contained in timer modules (`TimerModule` in Figure 4.1). As these modules follow the standard timer semantics defined in IEC 61131, we do not need an automata description for the semantics definition or for the implementation. However, the formal verification requires a consistent model describing both the non-timed and timed parts. The semantics of the PLCspecif timer modules are defined based on timed automata [BY04] in [*r*22]. Accordingly, the semantics of state machine modules can also be represented by timed automata, but without any clock variables. We defined these timed automata in a straightforward way that matches the previously used automaton semantics. This way we can have a consistent semantics for verification, but we do not have to face additional complexity in the implementation or if no timers are used.

Publications related to this section. The high-level ideas of the structure of PLCspecif were published first in [*e*19]. Later, [*c*12] provided a more detailed discussion of the specification language, focusing on the point of view of the users. The technical report [*r*22] provided a detailed syntax and formal specification definition.

4.4 Checking Invariant and Well-Formedness Properties on PLCspecif

Motivation. A formal specification language allows the users to write unambiguous, precise specifications. However, this does not mean that the specification will be correct, i.e. it matches the intentions of the specifier. In this section two different cases are considered.

- The base formalisms selected for PLCspecif, e.g. the state machines provide intuitive, easy-to-understand descriptions of certain behaviours. However, some aspects of the specification

might remain hidden. For example, the functionality may be well-represented by state machines, but invariant (safety) properties may remain hidden (e.g. two given output variables cannot have "true" values at the same time). These requirements can be additionally specified in PLCspecif, which can then be verified on the specification itself.

- Formal specifications are prone to human errors, in the same way as informal specifications. The main advantage of using formal methods is the possibility of mathematically sound analysis. For example, the satisfaction of various well-formedness rules can be checked. This can help to write correct specifications without contradictions, forbidden constructs or certain unintended behaviours.

The rest of this section discusses these two verification solutions and their applicability in more detail.

4.4.1 Verification of Invariant Properties

In line with the presentation in Section 4.3.1 each PLCspecif module can be extended with a set of invariant properties. The goal of these declarative requirement specifications is to complement the rather imperative function descriptions. If the specifier realises an invariant property to be respected (e.g. two given variables should not be "true" at the same time or an output value should be within certain limits) but based on the core logic definition it is not clear whether the property is always respected or not, it can be explicitly included in the description. To formalise these invariant properties, we can use requirement patterns as in Chapter 3.

It is not enough to define these requirements, their satisfaction should also be checked. Easy to observe that checking these invariant properties is similar to the verification problem in Chapter 3. The overview of this invariant checking workflow is shown in Figure 4.7. The only main difference is that the source of the verification model is not the implementation, but a specification [*c*5].

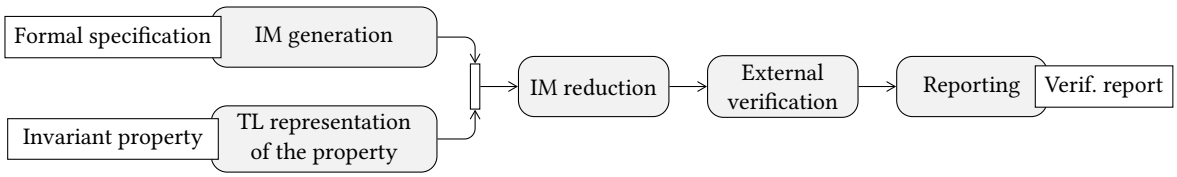


Figure 4.7: Overview of the PLC invariant checking workflow

Mapping PLCspecif specifications to IM. Section 4.3.4 discussed that the formal semantics definition of PLCspecif was designed in a way that it is close to the IM of the verification workflow, thus the PLCspecif specifications can also be represented by the intermediate model language.

The semantics of PLCspecif is formalised as a transformation to a timed automaton (as described in Appendix E (p. 145)). Clocks and invariants are used only to describe the PLC timers (TON, TOF, TP). An automaton corresponding to a PLCspecif specification without any timer modules can always be translated to an IM, because the rest of the elements in the timed automaton (e.g. locations, edges, variables) can systematically be translated to the corresponding elements (locations, transitions, variables) in IM. This systematic translation is described in Appendix E.3 (p. 152). If the IM is constructed by mapping the elements of the TA to the corresponding elements of the IM, then the behaviour of the IM and TA will be the same, as the semantics of the corresponding elements coincides on the level of state-transition graphs (this way having the same small-step operational semantics).

The automaton describing a PLCspecif module cannot have conflict between two transitions, therefore there could not be any semantic mismatch due to the different handling of non-determinism between the timed automata formalism defined for PLCspecif and the IM.

For verification purposes, we represent the timers used in PLCspecif specifications as proposed in Section 3.3.1. The same restrictions apply for the validity of the verification results as discussed before.

This allows to reuse the model checking workflow for invariant checking with no additional development effort besides the implementation of the "PLCspecif to IM" translation. However, as it will be discussed later in this chapter, other verification approaches (conformance checking) can also benefit from this IM representation. As the verification workflow is practically the same in this case too as in Chapter 3, we omit the further discussion of this method.

Example. *The example specification in Figure 4.5 contains an invariant property: "ALWAYS PMin $\leq$ Value $\leq$ PMax ASSUMING PMin $\leq$ PMax". This requirement seems to be obvious: the value* `_Value` *is assigned such that it is always within the limits specified by* `PMin` *and* `PMax`*. However, by verifying this property formally it can be shown that the property is not always satisfied. If the interval between* `PMin` *and* `PMax` *does not include 0, the requirement is violated when the module is disabled, as in that case the output* `Value` *will be set to zero.*

4.4.2 Static Analysis of Well-Formedness Rules

The formal abstract syntax definition of PLCspecif [*r*22] describes the metamodel of the specification language, but also a set of well-formedness constraints (currently 72 different rules). These rules describe additional constraints that are not expressed by the metamodel. Any valid specification shall respect all the well-formedness rules.

Most rules are simple, e.g. requiring name uniqueness, restricting data types, forbidding certain connections. These rules can be checked by traversing the abstract syntax tree of any specification and looking for violations of these constraints [*c*5].

Some constraints are more complex. For example, rule *WF-SM-8* in [*r*22] describes that the guards of the non-triggered transitions leaving a given state should be mutually exclusive, i.e. there should be at most one enabled transition leaving a given state. This ensures the lack of non-determinism. Another well-formedness constraint (*WF-SM-6*) describes that it should be forbidden to have an infinite ESF step, i.e. there should not be any infinite firing sequence of non-triggered transitions possible.

These more complex requirements could be checked using model checkers, similarly to Section 4.4.1. However, these rules can be formalised as Boolean satisfiability problems (SAT). For example, consider having a state s with outgoing non-triggered transitions $t_1, t_2, \ldots$. If these transitions have guards $g_1, g_2, \ldots$, respectively, then checking the well-formedness is equivalent to checking the satisfiability of $\bigvee_{\forall i,j\colon i \neq j} g_i \wedge g_j$. The satisfaction of this formula can be checked by a SAT solver, such as Z3 [MB08] that is used in our proof-of-concept implementation. If the given formula is satisfiable, the well-formedness constraint is violated. The "witness" (model) returned by the SAT solver gives an indication of the source of the problem (i.e. which transition pair is in conflict).

In this analysis it is assumed that all variables in the requirements are independent from each other. This allows the lightweight, quick check of the well-formedness constraints, but this might introduce false positives, reports of false well-formedness violations. If necessary, by representing the complete specification in IM and using the model checkers can provide a more precise answer, but typically it is more resource-consuming.

Publications related to this section. A brief overview of the verification methods related to PLCspecif was presented in [*c*12; *c*7]. More details are provided in [*c*5].

4.5 Code Generation[11]

Motivation. A complete, concrete behaviour specification language (such as PLCspecif) may be the source of automated implementation generation, which can greatly reduce the manual implementation effort. However, lowering the amount of manual work is only one of the motivations of providing a code generation method for the proposed specification language. An additional reason for that is to demonstrate that PLCspecif specifications are implementable (either manually or automatically). Furthermore, this feature can be a justification for the increased specification effort required by using a formal specification language (compared to the current informal specification practices), therefore this may foster the acceptance of formal specification.

Requirements. The *primary requirement* towards a code generation method is to provide *correct code*, more precisely an implementation whose behaviour corresponds to the behaviour described by the specification. In the process control domain, a *secondary*, but still important requirement is to generate an understandable code that can eventually be modified manually. In cases when the PLC program should be modified without stopping the PLC (e.g. urgent intervention), for technical reasons the whole application cannot be redeployed, therefore the expert developers shall be able to read and manually modify the code.

Example. *To illustrate the concepts of code generation, a simple R-S flip-flop component has been selected as an example. This flip-flop has two reset inputs and a set input. If one of the reset inputs is "true", the normal output (`Q`) of the component will become "false". If the `Set` input is "true", the output `Q` becomes "true". The outputs keep their state if there is no reset or set input. The reset input has priority over the set input. Besides the normal output `Q`, its negated value should be also produced (`notQ`)*[12]*. The PLCspecif specification of this component is presented in Figure 4.8.*

Both transitions in Figure 4.8 are event-triggered, therefore in one execution at most one of them can fire. Although there is a cycle in the state machine, an infinite firing sequence is not possible. In this example, there is no ESF step. The automaton constructed following the semantics definition for this example specification can be seen in Figure 4.9. As there is no ESF step in this example, the self-loops such as $\ell_1 \to \ell_1$ in Figure 4.6 are not present. The labels and colours correspond to Figure 4.6.

4.5.1 Overview of the Code Generation Method

Both the source and target languages (in our case PLCspecif and SCL code, respectively) have a concrete syntax, an abstract syntax and underlying semantics. The abstract syntax plays a central role, as typically the concrete syntax and the semantics of the language are both defined based on that. The *code generation problem* is to provide an implementation (i.e. source code) in concrete syntax for a given specification in concrete syntax having corresponding semantics. The link to be established is illustrated by the arrow *a)* in Figure 4.10. However, the implementation of the code generation is typically a model-to-text (M2T) transformation from the abstract syntax of the specification or model

[11] This section is an extended and adapted version of Section IV of [*c*7].

[12] Defining both `notQ` and `Q` seems to be redundant, but it illustrates the fact that PLC programs often define several similar outputs for convenience.

RSFlipFlop

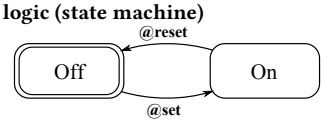
This module represents a flip-flop with two reset inputs and an edge-driven set input.

Assigned inputs:	**Assigned outputs:**
• AutoReset : BOOL • ManualReset : BOOL • Set : BOOL	• Q : BOOL • notQ : BOOL

Input signal definitions: — (none)

Event input definitions:
- @reset ⇐ AutoReset OR ManualReset (`pri`=1)
- @set ⇐ Set (`pri`=2)

Core logic (state machine)



Output signal definitions:
- Q = `in_state`(On)
- notQ = NOT `in_state`(On)

Invariant properties:
- ALWAYS Q ≠ notQ

Figure 4.8: R-S flip-flop module specification [*c7*]

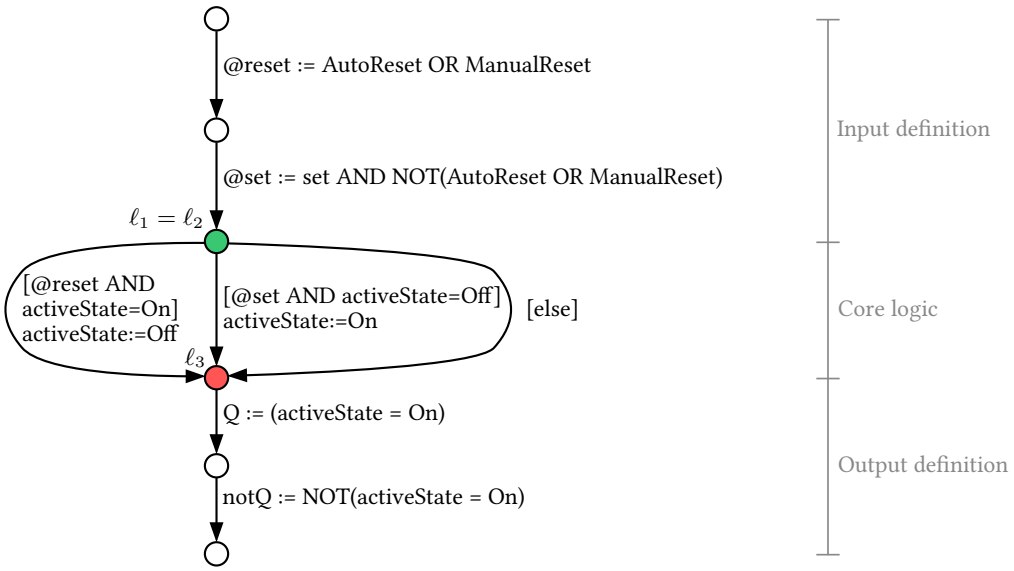


Figure 4.9: Automaton (extended with variables) describing the semantics of module RSFlipFlop [*c7*]

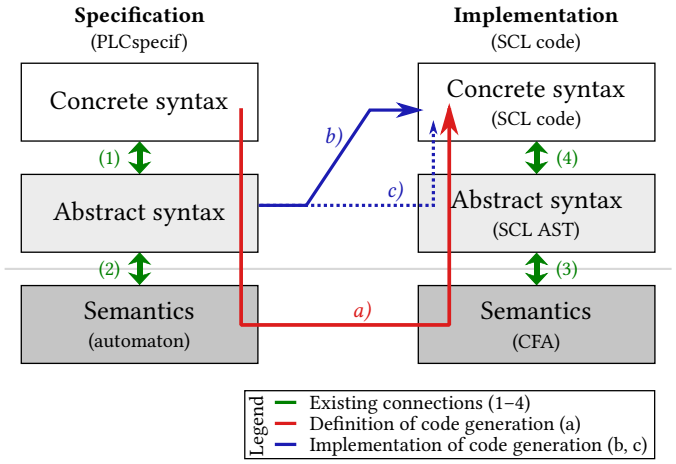


Figure 4.10: Overview of the definition and implementation of code generation (based on [c7])

to the concrete syntax of the implementation (arrow *b)* in Figure 4.10). To ensure that the semantics of the generated code matches the semantics of the specification, the transformation should be constructed based on the semantics definitions.
To provide the link *a)* of Figure 4.10, the following steps are needed (see also the green numbers and arrows in Figure 4.10).

1. The concrete syntax of the specification (given by the specified) is internally represented using its abstract syntax. This was briefly presented in Section 4.3.
2. The abstract syntax of the specification has to have well-defined semantics. In our case, the semantics of the specification is based on its translation into an automaton (extended with variables) which has a precise semantics. Section 4.3.4 already discussed this step.
3. The semantics of the implementation should correspond to the semantics of the specification. The automaton-based semantics of the specification is systematically mapped to the control flow automaton of the implementation, thus this is the basis of establishing the link. This step is described in Section 4.5.2.
4. The concrete and abstract syntax of the implementation should correspond to the control flow automaton semantics. This can be achieved as presented in Section 4.5.3.

4.5.2 Semantics Based on Control Flow Automata

We consider a control flow automaton as a generic low-level representation of program code, providing semantics for the programs.

Definition 4.1 (Control flow automaton, based on [Sut08]). A *control flow automaton* (CFA) is a tuple $CFA = \langle Q, q_0, q_{out}, V, \rightarrow \rangle$, where:

- Q is a finite set of locations,
- $q_0 \in Q$ is the initial (entry) location,
- $q_{out} \in Q$ is the exit location,
- V is the finite set of variables, and
- $\rightarrow \subseteq (Q \times G \times Q) \cup (Q \times VA \times Q)$ is the finite set of transitions.

Each transition is labelled by a guard $g \in G$ or a variable assignment $a \in A$. A guard is an expression evaluated on the variables to a Boolean value. A variable assignment determines the

> new values of the variables based on their current valuations. ▪

A CFA is similar to a control flow graph (CFG) [All70], but the operations label the edges instead of the nodes, thus the nodes of a CFA serve only as "junction points". Notice that a CFA cannot model certain aspects of the programs, such as pointers, recursion, threads [Sut08].

The automaton extended with guards and variable assignments, as the one used for the semantics description of PLCspecif is close to a control flow automaton defined above. The typical constructs used in the automaton can be mapped to CFA structures. The generated code is based on the code representation of these CFA constructs.

- The linear paths of the automaton (without any junction, e.g. the input and output definitions) can be represented as sequential variable assignments.
- The loops used for the ESF steps of the state machines can be realised using loops in the CFA (corresponding to `WHILE` loops in the abstract syntax tree (AST), where the exit condition of the `WHILE` loop is the guard of the one single transition that leaves the location with the loop). As the guards of the different loop edges in the automata should be mutually exclusive, their order (the order of the corresponding state changes) can be arbitrary.
- The parallel transitions representing event-triggered transitions of the state machine module can be represented using one or more conditional branches. The conditions of the automata transitions represent checking the actual state of the state machine, the active event input and the guard of the corresponding event-triggered transitions. The variable assignment connected to an automaton transition alters only the active state of the state machine. As the guards of these transitions are mutually exclusive, it does not matter in which order they are checked in the implementation.
- Timed transitions and clocks are used only in the timer modules. As they are defined to match the semantics of standard PLC timers, they can be represented directly with their standard implementation, their timed automata representation is only used for verification purposes.

4.5.3 Generating the Concrete Implementation

In the previous sections we described the main concepts of the transformation to the implementation. Section 4.5.2 presented which constructs of the concrete syntax could provide the required CFAs. However, some questions were left open and it is practical to make some modifications.

Representing enumerated types. PLCspecif and the underlying automata formalism support enumerated types for variables, e.g. `activeState` was defined with an enumerated type on all the basic states of the module. The programming languages of some PLC manufacturers do not support enumerated types, in these cases these variables can be implemented by an integer, or by n Booleans, with an 1-of-n encoding. It is also possible to give a more compact encoding using $\lceil \log_2 n \rceil$ Booleans.

Loop unfolding. The ESF step is essentially a loop until no more non-event-triggered transition can be fired. While it is straightforward to implement this as a while loop, the usage of loops is often regarded as bad practice in PLC programs since it is difficult to predict the execution time of a loop at compilation time. The ESF steps are expected to be short, as they typically represent error-handling related behaviour (e.g. maintaining a "healthy state"). Therefore we unfold these ESF steps, explicitly representing each possible firing sequence. This allows the demonstration of the absence of infinite loops.

```
FUNCTION_BLOCK RS_FF
    VAR_INPUT
        AutoReset : BOOL; // Reset request from logic
        ManualReset : BOOL; // Reset request from operator
        Set : BOOL; // Set request
    END_VAR
    VAR_OUTPUT
        Q : BOOL; // Normal (positive) output
        notQ : BOOL; // Negated output
    END_VAR
  // Events
    VAR
        _E_reset : BOOL; // Event Resets the RS flip-flop.
        _E_set : BOOL; // Event Sets the RS flip-flop.
    END_VAR
  // State variables
    VAR
        s_Off : BOOL := TRUE;
        s_On : BOOL;
    END_VAR

  // RS_FF (RSFlipFlop)
  // ==========================================
  // This module represents a flip-flop with two reset inputs and an edge-driven set input.
  // ------------------------
  // Events
    _E_reset := (AutoReset OR ManualReset); // Event reset (pri=1)
    _E_set := Set AND NOT _E_reset; // Event set (pri=2)

    IF _E_reset AND s_On THEN // Transition tReset
        s_On := FALSE; s_Off := TRUE;
    END_IF;
    IF _E_set AND s_Off THEN  // Transition tSet
        s_Off := FALSE; s_On := TRUE;
    END_IF;

  // Outputs
    Q := s_On;
    notQ := NOT (s_On);
  // End of RS_FF
END_FUNCTION_BLOCK
```

Listing 4.1: Code generated from the R-S flip-flop example specification

Event-triggered transitions. To obtain a CFA that exactly matches the automaton semantics, the implementation of firing the (at most one) event-triggered transition between the two ESF steps would consist of one single `IF-THEN-ELSIF-ELSE` block. A big monolithic block like this can be difficult to read and understand. Instead, we split it into several conditional blocks based on the triggering events. As at most one event can be active in an execution, this does not modify the semantics of the implementation.

The code generated for the R-S flip-flop example (Figure 4.9) can be found in Listing 4.1. The variables starting with `s_` represent the currently active state in a 1-of-2 encoding. The variables starting with `_E_` represent the currently active (at most one) event input.

4.5.4 Providing Readable Code

The code generation based on the principles discussed before satisfies the primary (correctness) requirement. However, it did not take the requirement of providing readable code into account. To tackle this problem, this secondary requirement should be further refined. The code generated based on PLCspecif provides the following features to fulfil the secondary requirement.

- **Readability.** The code is indented correctly and uses whitespaces consistently.
- **Understandability.** The code is easily understandable. It follows a general structure similar to the specifications' structure, it is formatted to support the meaning (e.g. expressions are well-formatted, an expression described by an AND/OR table in the specification follows a similar structure in the implementation too).
- **Configurability.** The developer is able to configure the way of implementation, where the used solution can be chosen from a set of possible ones (e.g. representing an enumeration using an `INT` or several `BOOL` variables).
- **Simplicity.** The code should be regular, but as simple as reasonably possible. For example, redundant variables should be avoided. Furthermore, unnecessary constructs, such as always true expressions (tautologies) and unnecessary `if` statements should be avoided, trivial nested conditional statements should be flattened, etc. This can be achieved by different configuration options and by using simplification algorithms on the implementation (cf. Section 4.5.5).

  Example. *In case of Listing 4.1, it is unnecessary to have two separated variables for `s_On` and `Q`. In some cases, merging these variables could help the user to easier understand the code, and it also reduces the memory consumption. In other cases, this could have the opposite effect, causing difficulty in understanding. Therefore the user is able to configure these options.*

4.5.5 Generation Process

Most of these requirements towards generating readable code can be satisfied by having a *model-to-text* (M2T) transformation. The input of the transformation is the specification in abstract syntax, and the output is implementation in concrete syntax, in our case the SCL code representation (represented by arrow *b)* in Figure 4.10). This is the workflow we have chosen for our current, proof-of-concept code generator.

However, some advanced features might benefit from a more complex workflow. For example, merging the variables with the same meaning or simplification of conditional statements can make the simple M2T code generator much more complex, thus more error-prone. Decoupling the simplifications from the core code generation features may solve this issue, but then a new, three-step workflow is necessary. First, the AST of the implementation is generated (model-to-model transformation). Then the simplifications can be made on this AST, and finally the AST is translated to the real implementation using a M2T transformation. This is represented by arrow *c)* in Figure 4.10.

Decoupling the two transformations can also help to solve the portability problem of PLCs. Even if the IEC 61131-3 standard defines the syntax of the PLC programming languages, each manufacturer has slightly different variants of the programming languages, making it difficult to change the hardware supplier. By having a manufacturer-independent abstract syntax and a manufacturer-dependent M2T transformation to generate the program code, the problem of vendor-incompatibility could be reduced. Currently this three-step generation method is a future work.

Publications related to this section. The applicability of PLCspecif in code generation was briefly discussed in [*c*12], then later a detailed description was provided in [*c*7].

4.6 Conformance Relations and Conformance Checking[13]

A correct code generation tool ensures that the behaviour of the implementation generated matches the specified behaviour. However, using only generated code is not always feasible. Due to the typically long lifespan of the industrial control systems it is unavoidable to support legacy systems. Furthermore, code generation may be forbidden for certain cases, for example in case of safety-critical systems based on Siemens PLCs (see Section 3.6). For these cases an additional analysis solution has to be provided which can assess the conformance between the implementation and the specification.

Various *behavioural equivalence relations* were introduced in the past for these purposes, such as the widely-known trace equivalence, (bi)simulation, and ioco relations [Bac81; Par81; Tre96]. For reactive systems [HP85] these relations are useful in practice, but this is not always the case for ICS. A PLC with a typical cycle time of 1–100 ms may control a slow process (e.g. a cryogenic plant) where certain responses of the plant are expected within minutes or even later. For many signals, a delay by 1–100 ms has no significant impact. Furthermore, these slight changes of the outputs, often caused by code reorganisation, cannot be easily avoided in a complex application. These *acceptable differences* appear as false positives in the equivalence checking. As typically only a single counterexample is provided, it is difficult to identify and exclude all these cases.

Our key observation is that the widely-known conformance relations and even the PLC-specific conformance relations (e.g. in [Ule+15]) are typically too strict. The ICS domain needs conformance relations with configurable sensitivity to be useful in the development process. In summary, *development of new conformance relations specifically targeting the PLC-based industrial control software is needed to make conformance checking useful in practice.*

The rest of this section introduces new conformance relations with different levels of permissivity, which are suitable for the control software development domain. The methods to check the satisfaction of these methods will also be provided in this section.

4.6.1 Domain Requirements

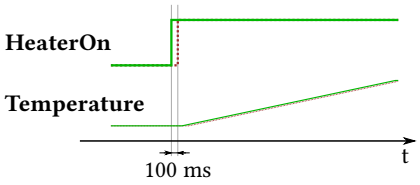
This section overviews the special needs[14] of the industrial control systems domain to provide motivation for our work.

Use cases. We consider three main cases of applying conformance checking:

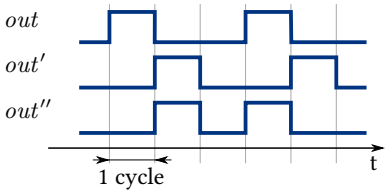
- **Specification–implementation.** An implementation and the corresponding specification can be compared to check the conformance of a manually developed implementation and the specification. Comparing a specification and an implementation can also be useful if the implementation is generated, but the code had to be manually altered. After performing the modifications on the specification too, the consistency can be re-established between the implementation and the specification.
- **Implementation–implementation.** For instance, two implementations can be compared to check different implementations of the same specification, or to check that an extended version of an implementation still provides the previous behaviour (in addition to some new behaviours).

[13] This section is an adapted version of [*c*6].

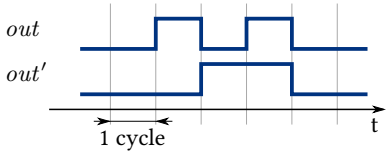
[14] In this section, "needs" will always refer to the requirements for conformance relations.



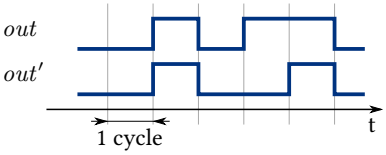
(a) A delay of one PLC cycle often causes an unnoticeable difference.



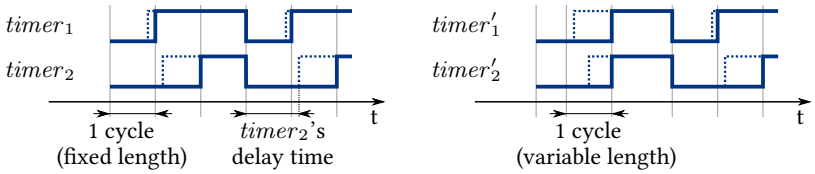
(b) The allowed delays can be constant (same for the whole execution, out') or varying (out'').



(c) The varying delay may modify the amount of rising/falling edges.



(d) For some signals the edge number preservation is more important than the pulse duration preservation.



(e) The fixed PLC cycle length can hide some behaviours. The timeouts (rising edges) of $timer_1$ and $timer_2$ never happen in the same cycle using fixed lengths, but it is possible using variable cycle length ($timer'_1$ and $timer'_2$).

Figure 4.11: Motivational examples [*c*6]

- **Specification–specification.** For example, two specifications can be compared to be sure that the new version of the specification preserves the behaviour given by the previous version, or an optimised/reorganised specification still behaves in the same way as the previous version.

In the following, we focus mainly on the specification–implementation use case.

Permissibility of the relations. We have extracted some motivational examples from CERN's PLC-based control systems (Figure 4.11). The main observation is the fact that small differences of the control outputs may be observable, but practically equivalent from the point of view of the properties of the control functions in certain cases (Figure 4.11(a)). Accordingly, if those differences are identified as problems, they are often treated as false positives by the developers. This might reduce the applicability and acceptance of a formal verification method. Focusing on the relevant differences implies the need for *permissive conformance relations*. However, this need co-exists with the need for strict equivalence, depending on the role of the outputs. Often some outputs are insensitive to small delays (e.g. an output used only for information by the supervision system or controlling slow processes), while others might not allow any differences. Therefore the conformance relations should not be selected for two compared artefacts (specification or implementation), but for pairs of outputs (one output per compared artefact).

The level of permissibility is given separately for each output pair, and not included in the specification. This way the specification can be kept "clean", describing the ideal required behaviour. Also,

this makes the comparison of two implementations possible. Being able to select different relations for different output pairs also gives flexibility in checking artefacts (specification or implementation) with different input/output signatures. For instance, the behaviour of an artefact M can be compared to M', which is M extended with new outputs. By selecting conformance relations only for the variable pairs contained in both M and M' it is possible to analyse if the extension had any side-effect on the base behaviour. However, this does not mean that only one variable pair's conformance can be checked in a single model checking run, the checks for several variable pairs can be conjuncted.

Even if an output may allow some delay, in some cases the amount of allowed delay should be fixed during the whole execution (e.g. *out*′ compared to *out* in Figure 4.11(b)), in other cases the delay may vary during an execution (e.g. *out*′′ compared to *out* in Figure 4.11(b)). The allowed differences depend on the usage of the signal. Moreover, for some signals, the sum of pulse durations should be equal to call them conformant (Figure 4.11(c)), for some other signals the pulse duration is secondary, but the number of rising or falling edges should be the same (Figure 4.11(d)).

Based on the discussion above, the following three main conformance relation categories are defined, depending on the permissibility of the conformance.

- **Strict equivalence relation.** This relation requires the same output sequences for the same input sequences under the same timing conditions in the two artefacts (specification or implementation).
- **Permissive conformance relations with fixed delay.** These relations permit delays by certain number of PLC cycles between the outputs of the two artefacts for the same inputs and timing, however the delay should be constant, therefore the shape of the two output signals will be practically identical.
- **Permissive conformance relations with variable delay.** These relations also allow delays in the outputs, and the amount of delay may vary during the execution (within the defined limits). In certain cases such relation provides enough restriction for the conformance checking, e.g. in case of status outputs used for informative purposes in the supervision systems.

Cycle time. Although PLCs may have interrupts, interrupt-driven reactive behaviour seems to be uncommon in practice. Here we do not consider user-defined interrupts and we treat PLCs as transformational systems [HP85], transforming an input sequence to an output sequence under some timing conditions. However, e.g. communication- or operating system-related interrupts may occur during execution. A side effect of them is that the length of a scan cycle may vary in most of the PLCs. This causes non-determinism in the otherwise deterministic execution. As the PLC timers' delay parameters are defined in physical time units, not in number of cycles, this can have observable collateral effects (Figure 4.11(e)). The length of a scan cycle also depends on the number and type of executed instructions, or on the used hardware. Accordingly, we consider variable cycle lengths (cycle durations) during the execution. However, we assume that the two compared artefacts process the same sequence of inputs (otherwise they will be trivially inequivalent). To assure this, during equivalence checking we consider the same input and cycle length sequences for both artefacts. Note that programs running on non-fail-safe PLCs typically do not have fixed cycle length, therefore the implementation should not rely on the exact value of the cycle length (this is typically treated as non-deterministic by the developers), therefore these assumptions should not pose any restriction.

4.6.2 Conformance Relations

This section introduces the conformance relations defined upon the identified needs. Section 4.6.2.1 describes the modelling and formalisation of the artefacts and conformance checking problem. Section 4.6.2.2 defines the conformance relations.

In this section the usage of PLCspecif specification will be assumed, but the defined conformance relations are more generally applicable to PLC-based control systems and they could be used with other formal specification languages too.

4.6.2.1 Formalisation and Assumptions

For the formal definition of conformance relations, we provide the formalisation of behaviour semantics of PLC programs and we introduce the necessary symbols.

Time assumptions. As mentioned previously, a good compromise had to be found between precision and performance of verification for time modelling. The assumption that the PLC cycle length is fixed is too vague (cf. Figure 4.11(e)), and it may hide possible behaviours. On the other hand, if the time is modelled precisely, checking the conformance relations becomes troublesome. We follow an approach similar to the realistic approach discussed in Section 3.3.1: the length of each PLC cycle is non-deterministic with an accuracy of 1 ms (same as the accuracy of the PLC timers in most implementations). We do not model a finer granularity of time, i.e. the global clock does not advance during a cycle in our models. In other words, our PLC cycle model has two phases: a non-deterministic delay without any computation and the real execution modelled with execution time 0.

Generic modelling for different artefacts. Section 4.6.1 presented different use cases, involving checking of specifications and implementations in arbitrary combinations. As all combinations of artefacts might be compared, it is useful to have a common representation for both specifications and implementations. Based on the discussions in Chapter 3 and Section 4.4 it can be seen that the intermediate model language can provide a representation both for implementation and specification artefacts for verification purposes. As using the IM it is also possible to benefit from the reductions and model checker integrations already implemented in PLCverif, we will use the IM as common verification representation for conformance checking in the following. From this point, we often do not distinguish between implementation and specification, as typically they are represented in the same way and can be used interchangeably. In this case, we refer to both as *artefact* and we denote them by M or M'.

Formalisation of the semantics. To be able to formally define the conformance relations, first the model and behaviour of PLC artefacts should be defined. We do this in a generic way, covering both specifications and implementations.

> **Definition 4.2 (Model of a PLC artefact).** The model of a PLC artefact is a structure $M = \langle V_I, V_O, V_L, \mathit{val}, \pi, s_0 \rangle$, where V_I is the set of input variables, V_O is the set of output variables, V_L is the set of internal variables, $\mathit{val}\colon (V_I \cup V_O \cup V_L) \to 2^{\mathit{Val}}$ is the set of possible values for each variable (where *Val* is the set of all representable values, typically Boolean and bounded integer values in PLCs), π is the behaviour description (source code or specification) of the artefact with an initial state s_0. ▪

Let us denote the input value space[15] by I, the output value space by O, the internal state space by S (similarly to [Bec+15]) and the set of potential cycle lengths in ms by $T = \{1, \dots \tau\}$ (where τ is a configurable upper limit on the cycle length, enforced by the PLC's watchdog). Then I^ω is the set of possible infinite input sequences. $\underline{i} = (i_1, i_2, \dots) \in I^\omega$ denotes an infinite input sequence, where each i_j is a vector assigning values for each input variable $v \in V_I$ of the artefact, thus can also be considered as a function $V_I \to \mathit{Val}\ (\forall v, i_j : i_j(v) \in \mathit{Val}(v))$. The definitions of O^ω, $\underline{o}$, T^ω and $\underline{t}$ are similar. The vector i_j represents the input values observed (sampled) at the beginning of cycle j, which has a length of t_j and after this t_j time will provide outputs as defined in o_j. Based on these symbols, the trace semantics of a PLC artefact can be drawn up.

Definition 4.3 (Semantics of a PLC artefact, based on [Bec+15]). The *observable* behaviour b_M of a PLC artefact M is the function $b_M \colon I^\omega \times T^\omega \to O^\omega$, defined by (π, s_0). ▪

Notice that behaviour description (π, s_0) contain more information than the observable behaviour b_M. Furthermore, b_M is a simplified view of the real behaviour for conformance checking purposes, due to the way of time modelling.

4.6.2.2 Definition of Conformance Relations

In this section we define in total six conformance relations ($\mathsf{pconf}_1, \dots, \mathsf{pconf}_6$) in three categories, with different levels of permissibility, reflecting to the previously discussed needs. We recall that different output variables might need different levels of conformance: for some outputs we may require strict equivalence, while some other outputs may be allowed to be delayed. Therefore each of the conformance relations targets the conformance of two corresponding output variables, not whole modules. They can also be separately parametrised.

To keep the formal definitions short, the following symbols are defined: $\underline{w} \triangleq \Pi_v(b_M(\underline{i}, \underline{t}))$ and $\underline{w}' \triangleq \Pi_{v'}(b_{M'}(\underline{i}, \underline{t}))$, where Π_v denotes projection of the behaviour of the PLC artefact to variable v, i.e. $\underline{w}$ is the sequence of values of output variable v given by M for the sequences $\underline{i}, \underline{t}$[16]. We assume that variables v and v' are corresponding to each other in M and M', respectively, thus the set of their possible values are the same ($\mathit{val}(v) = \mathit{val}(v')$).

Strict equivalence. Strict equivalence ($M\ \mathsf{pconf}_1^{v,v'}\ M'$) is the simplest and strictest conformance relation between two variables of two artefacts defined here. It is satisfied if the two artefacts, M and M', assign always the same value to the output variables v and v', if the same input sequence is provided under the same timing conditions. This relation is similar to the perfect equivalence relation of [Bec+15], with the previously discussed time extensions.

Definition 4.4 (pconf_1). $M\ \mathsf{pconf}_1^{v,v'}\ M' \iff \forall \underline{i} \in I^\omega, \underline{t} \in T^\omega \colon \underline{w} = \underline{w}'$. ▪

Permissive conformance relations with fixed delay. As mentioned previously, the fixed permissive conformance relations allow delays between the corresponding outputs of the two compared artefacts. More precisely, $M\ \mathsf{pconf}_2^{v,v',n} M'$ is satisfied, if the delay of output variable v' compared to v is exactly $n \in \mathbb{Z}$ cycles. If n is positive, the output v' of M' is delayed compared to the output v of M.

[15] Each element of I assigns a value for each input variable of the artefact.

[16] Notice that $\underline{w}$ is a function of $\underline{i}, \underline{t}, v, M$ and $\underline{w}'$ is a function of $\underline{i}, \underline{t}, v', M'$. However, we omit to denote this in the following to keep the definitions compact and readable.

For the formalisation the following notation is introduced. If $\underline{w} = (w_1, w_2, w_3, \dots)$ is the output sequence of variable v, then $\underline{w}_{-1} = (*, w_1, w_2, \dots)$, i.e. $\underline{w}_{-1}$ will correspond to the output sequence of the variable v delayed by one cycle. The symbol $*$ denotes a "do not care" value. Any equality should be evaluated to true that contains a $*$ (e.g. $2 = *$ is true). More generally: $\underline{w}_{-n} = (\underbrace{*, \dots, *}_{n}, w_1, w_2, \dots)$ and $\underline{w}_{+n} = (w_{1+n}, w_{2+n}, \dots)$. Notice that $\underline{w} = \underline{w'}_{+n} \iff \underline{w}_{-n} = \underline{w'}$.

Definition 4.5 (pconf$_2$). $M \ \mathsf{pconf}_2^{v,v',n} \ M' \iff \forall \underline{i} \in I^\omega, \underline{t} \in T^\omega : \underline{w} = \underline{w'}_{-n}$. ▪

Example. *In Figure 4.11(b), the out and out′ signals do not violate* $M \ \mathsf{pconf}_2^{out,out',1} \ M'$ *(assuming that out′ keeps the same constant shift compared to out infinitely), but* $M \ \mathsf{pconf}_2^{out,out'',1} \ M'$ *cannot be satisfied (no matter what is the rest of the signal).*

The relation $\mathsf{pconf}_3^{v,v',K}$ generalises $\mathsf{pconf}_2^{v,v',n}$. The parameter K of $\mathsf{pconf}_3^{v,v',K}$ is a set $K \in 2^{\mathbb{Z}}$ instead of a single number. $M \ \mathsf{pconf}_3^{v,v',K} M'$ iff $M \ \mathsf{pconf}_2^{v,v',n} M'$ holds for an $n \in K$. Using these notations, the formal definition of this conformance relation is the following.

Definition 4.6 (pconf$_3$). $M \ \mathsf{pconf}_3^{v,v',K} \ M' \iff \exists n \in K : M \ \mathsf{pconf}_2^{v,v',n} \ M'$. ▪

Permissive conformance relations with variable delay. Certain modifications in the specification or the implementation cause shifts only in certain cycles of the execution, not consistently (see Figure 4.11(b) for example). If these are acceptable differences, the previously defined conformance relations are still too strict. Therefore we introduce a relation allowing variable shifts in the output sequences.

The key idea of $\mathsf{pconf}_4^{v,v',K}$ is the following. Given the two output sequences $\underline{w} = (w_1, w_2, \dots)$ and $\underline{w'} = (w'_1, w'_2, \dots)$, and a set of allowed delays $K = \{k_1, k_2, \dots\}$, the two sequences are considered as conformant if for each w_i there is a $w'_j = w_i$ in i's K-surrounding, i.e. $w_i = w'_{i+k_1} \vee w_i = w'_{i+k_2} \vee \dots$; and for each w'_i there is a $w_j = w'_i$ in i's "$-K$"-surrounding (where $-K$ is the element-wise negation of K).

In the formal definition we use $\varphi(\underline{w}, \underline{w'}, K)$ which stands for "for each w_i there is a $w'_j = w_i$ in i's K-surrounding". Formally (for infinite vectors):

$$\varphi(\underline{w}, \underline{w'}, K) \iff \forall i = 1, \dots : \exists s \in K : 1 \leq i + s \wedge w_i = w'_{i+s}.$$

An illustration of φ can be seen in Figure 4.12. It shows that for the given example vectors $\underline{u}, \underline{u'}$ the $\varphi(\underline{u}, \underline{u'}, \{-1, 0, 1, 2\})$ is not satisfied because there is no match for $u_6 = 1$ in $\underline{u'}$. However, $\varphi(\underline{u'}, \underline{u}, \{-2, -1, 0, 1\})$ is satisfied for the same vectors, as there is a match for each u'_i in $\underline{u}$ (indicated by the arrows in Figure 4.12). The shaded area illustrates the range in which the corresponding value is searched, e.g. in the first example, u_2 should equal to any of $u'_1, \dots, u'_4$ to satisfy the relation φ.

Definition 4.7 (pconf$_4$). $M \ \mathsf{pconf}_4^{v,v',K} \ M' \iff \forall \underline{i} \in I^\omega, \underline{t} \in T^\omega : \varphi(\underline{w}, \underline{w'}, K) \wedge \varphi(\underline{w'}, \underline{w}, -K)$. ▪

Example. *The signals in Figure 4.11(b) do not violate* $M \ \mathsf{pconf}_4^{out,out',\{-1,0,1\}} \ M'$*, neither* $M \ \mathsf{pconf}_4^{out,out'',\{-1,0,1\}} \ M'$*, assuming that the shown signals are repeated infinitely. It is worth noting that to argue about the satisfaction of these relations, the output sequences should be compared for any possible input sequence.*

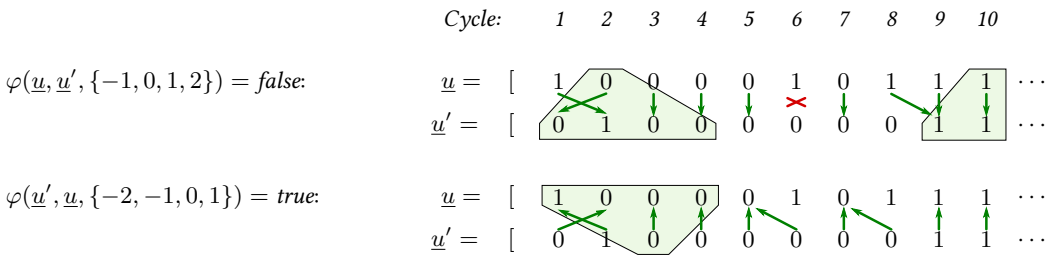


Figure 4.12: Example of relation φ (based on [c6])

Following the ideas described in Section 4.6.1, we define two additional restrictions to $\mathsf{pconf}_4^{v,v',K}$. The relation $\mathsf{pconf}_5^{v,v',K}$ requires *total pulse duration preservation* in addition to $\mathsf{pconf}_4^{v,v',K}$, thus for each allowed value of v their number of occurrences should be the same for the outputs of the two artefacts for every possible execution. More precisely, for infinite many finite prefixes of the output sequences of v and v', the difference in the occurrence numbers of each allowed value is zero.

The number of occurrences of value e in vector $\underline{w}$ is denoted by $\underline{w}\#e$ in the formal definition. Let us introduce also $\underline{w}_{[a,b]} \triangleq (w_a, \ldots, w_b)$ $(b \geq a)$, i.e. $\underline{w}_{[1,n]}$ is the n-long prefix of the vector $\underline{w}$.

Definition 4.8 (pconf_5). $M\ \mathsf{pconf}_5^{v,v',K}\ M' \Leftrightarrow M\ \mathsf{pconf}_4^{v,v',K}\ M' \wedge \forall \underline{i} \in I^\omega, \underline{t} \in T^\omega$: *for infinitely many* $n \in \{1, 2, \ldots\}, \forall e \in \mathit{val}(v) : \underline{w}_{[1,n]}\#e = \underline{w}'_{[1,n]}\#e$. ▪

Example. *$M\ \mathsf{pconf}_5^{out,out',\{-1,0,1\}}\ M'$ is not violated by the output pair shown in Figure 4.11(c) (assuming that the same signal shape is repeated infinitely), but $M\ \mathsf{pconf}_5^{out,out',\{-1,0,1\}}\ M'$ is violated in Figure 4.11(d) with the same assumption, as the total pulse duration for out' is shorter than for out.*

In other cases, the total pulse duration is not important, but differences in the number of rising or falling edges might have an impact on the behaviour. The relation $\mathsf{pconf}_6^{v,v',K}$ reflects the common usage of edge-driven signals, thus it is only defined for Boolean variables. The additional restriction of $\mathsf{pconf}_6^{v,v',K}$ compared to $\mathsf{pconf}_4^{v,v',K}$ is that the *number of rising edges* should be the same in the two outputs for every possible execution. To formalise this, we introduce the rising edge vector $\uparrow(\underline{w}) = (e_1, e_2, \ldots)$, where e_i is true iff $i > 1$, $w_{i-1} = \mathit{false}$ and $w_i = \mathit{true}$.

Definition 4.9 (pconf_6). $M\ \mathsf{pconf}_6^{v,v',K}\ M' \Leftrightarrow M\ \mathsf{pconf}_4^{v,v',K}\ M' \wedge \forall \underline{i} \in I^\omega, \underline{t} \in T^\omega$: *for infinitely many* $n \in \{1, 2, \ldots\} : \uparrow(\underline{w}_{[1,n]})\#\mathit{true} = \uparrow(\underline{w}'_{[1,n]})\#\mathit{true}$. ▪

Example. *In Figure 4.11(c), $M\ \mathsf{pconf}_5^{out,out',\{-1,0,1\}}\ M'$ is violated, as out' has fewer rising edges than out (with the same assumptions as in the previous examples). However, in Figure 4.11(d), $M\ \mathsf{pconf}_5^{out,out',\{-1,0,1\}}\ M'$ is not violated, assuming that the same signal is repeated infinitely.*

Notice that we focus on checking whether the two compared artefacts are conformant given the same inputs and cycle lengths. Consequently, we cannot show differences between two programs implementing the same specification with different complexity, thus with different execution times. A stricter relation could show these differences, but it could even mark any code non-conformant with itself, as on different hardware the execution time is different. We use a higher abstraction level, not taking the internal interrupts and hardware differences into account. In practice, the developers do likewise: if some deterministic, physical time conditions have to be satisfied, PLC timers are used, or

a fixed cycle time can be assumed in a fail-safe PLC. On the other hand, the relations introduced here can show the differences when the two artefacts give different outputs for the same inputs (wrong functionality) or when the delay between the two corresponding outputs (in number of cycles) is out of the given acceptable range.

4.6.3 Checking the PLC Conformance Relations

Obviously, to make the conformance checking useful in practice it is not enough to define the conformance relations. A method has to be established to check if two artefacts are conformant or not. Section 4.6.3.1 overviews the proposed conformance checking approach. It consists of generating models (detailed in Section 4.6.3.2) and temporal logic (TL) expressions (detailed in Section 4.6.3.3), then evaluating the conformance using a model checker.

It has to be noted that checking the conformance could be done based on other approaches as well, for example test cases could be generated to evaluate the correspondence between the implementation and the specification. However, for the advantages of model checking that were already discussed, model checking was chosen as the underlying technique to check conformance.

4.6.3.1 Overview of the Conformance Checking Approach

We recall the primary needs affecting the implementation of conformance checking: the method should be generic to support different use cases, thus different artefacts, and it should be able to scale up to large input artefacts.

The previously introduced verification workflow (described in Section 3.4) and the PLCverif tool implementing it already provide a solution for efficient model checking – using requirement-specific reductions – of PLC programs through the intermediate model language. It was already discussed in Section 4.3.4 that the PLCspecif specifications can also be represented by IM for verification. We have decided to reuse and to build on these solutions, thus to use model checking for conformance checking. An advantage of reusing the PLCverif intermediate models is that we can benefit from its built-in model reductions [*c*15]. The model (and the TL expression) can also be translated to the concrete syntax of various model checkers by PLCverif, thus we can use different model checkers for conformance checking.

To reuse the workflow of PLCverif, the following steps have to be executed:

1. *Generating* (compatible) *verification models* representing the two artefacts to be compared,
2. *Building composite verification models* for conformance checking,
3. *Generating temporal logic expressions* representing the conformance criteria (i.e. the conformance relation is valid between the two models iff the temporal logic expressions are satisfied on the composite verification model), and
4. *Performing model checking* using the composite verification models and the generated temporal logic expressions.

The Steps 1 and 4 are already supported by PLCspecif and PLCverif as discussed in Chapter 3 and Section 4.4. The composite model construction (Step 2) will be discussed in Section 4.6.3.2, the temporal logic expression representation of the conformance relations (Step 3) will be presented in Section 4.6.3.3. The complete conformance checking workflow is illustrated in Figure 4.13.

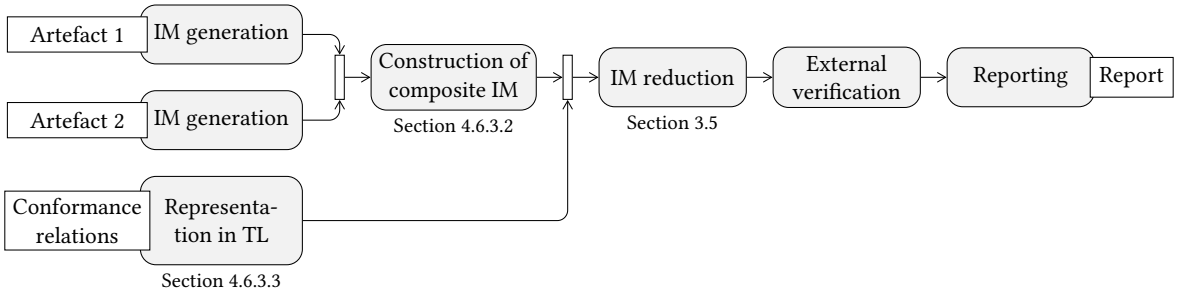


Figure 4.13: Overview of the proposed conformance checking workflow

4.6.3.2 Composite Model Generation

As mentioned before, the applied toolchain can produce models separately for the compared artefacts M and M'. The additional task is to generate a *composite verification model* $\Gamma_{M,M'}$ on which the evaluation of a TL expression can decide the satisfaction of the selected conformance relations.

First trials showed that in case of certain model checkers (e.g. nuXmv) the performance may significantly drop when complex temporal logic expressions are checked. Similar observation was already made for mode selection in Section 3.5.1. Therefore the model $\Gamma_{M,M'}$ is constructed in a way that it contains some parts of the conformance criteria directly, as described in the following.

- The corresponding input variables (identified manually or using heuristics) are merged: one of them is deleted and the other is used in both M and M'[17] (see Figure 4.14(a)).
- The cycle time representation (sequence of cycle times) is merged in a similar way (M and M' will have the same, non-deterministic PLC cycle length, given by $\underline{t}$).
- If the TL expression refers to delayed variables, i.e. it uses not only the current, but a previous cycle's value of a certain variable (cf. Section 4.6.3.3), the delayed variables are also included as new output variables in $\Gamma_{M,M'}$, as illustrated in Figure 4.14(b). v_{-1} represents a variable whose value equals to the value of v in the previous cycle.
- For $\mathsf{pconf}_5^{v,v',K}$ and $\mathsf{pconf}_6^{v,v',K}$ some other helper variables (P^v, Q^v) are defined (see later, in Section 4.6.3.3).

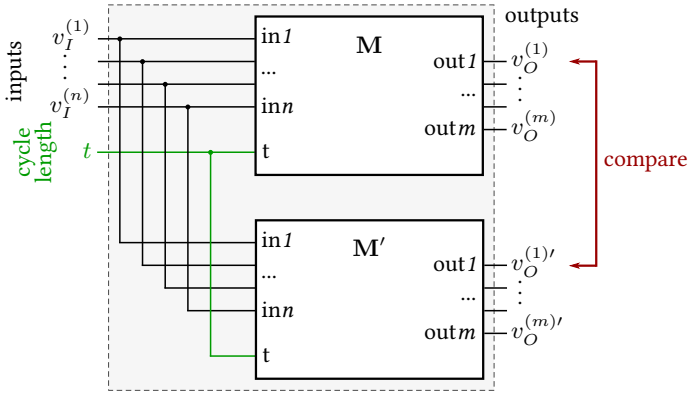
4.6.3.3 Temporal Logic Expression Generation

The next task is to represent the required conformance relations in TL according to the definitions of the relations. As $\Gamma_{M,M'}$ is constructed in a way that the two compared artefacts have the same inputs and timing conditions (that is required by every conformance relation), the TL expressions need to express only the comparison of outputs.

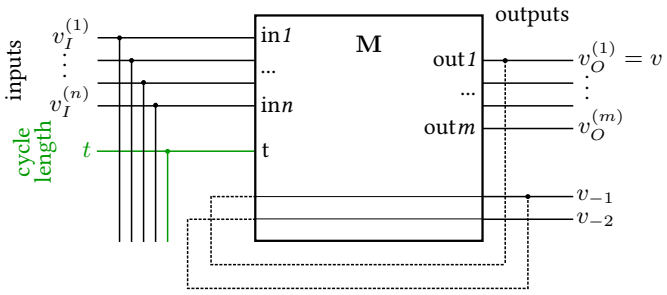
Here we overview briefly the ideas behind each of the six relations' TL representation. The formal definitions can be found in Table 4.2.

- The representation of the strict equivalence relation ($\mathsf{pconf}_1^{v,v'}$) is trivial, only the values of the two corresponding output variables (v, v') have to be compared.
- To check $\mathsf{pconf}_2^{v,v',n}$, the shifted values should be taken into account too, thus additional output variables should be generated during the construction of $\Gamma_{M,M'}$, as discussed in Section 4.6.3.2. It has to be noted that the outputs can only be delayed, the future values (e.g. v_{+1}) should not be referred in the TL expressions.

[17] This implies the assumption that variables representing physical inputs are not modified by the user program.



(a) Structure of the composite verification model $\Gamma_{M,M'}$



(b) Representation of delayed variables

Figure 4.14: Modelling methods [c7]

- To check $\mathsf{pconf}_3^{v,v',K}$ it is enough to check if $\mathsf{pconf}_2^{v,v',n}$ holds for at least one $n \in K$.
- The main idea of checking $\mathsf{pconf}_4^{v,v',K}$ is already introduced by defining the function φ in Section 4.6.2.2. However, this could cause looking ahead in time (see Figure 4.12). To avoid this and the complex TL expressions imposed by looking ahead, we determine the largest look-aheads (denoted by $\mu(K)$ and $\mu(-K)$) and we shift all comparisons towards the past by these amounts of cycles.
- Checking relation $\mathsf{pconf}_5^{v,v',K}$ requires additionally to check the different values of the output variables. In the model generation phase we automatically introduce a variable P^v for Boolean variables in the verification model $\Gamma_{M,M'}$. This variable is incremented by one if v is "true" at the end of a cycle and decremented by one if v' is "true" at the end of the cycle. Therefore $P^v = 0$ means that v and v' were "true" for the same amount of cycles. The additional requirement is expressed as P^v has to go back to zero always in the future.
  Checking this relation for non-Boolean variables can also be achieved, by a simple extension of the verification model. Let us add a new variable S^v to $\Gamma_{M,M'}$ with the same type as v. The value of S^v should be non-deterministic and this value should be kept during an execution. In this way the previously introduced P^v can count the differences in how many times v and v' had the value S^v. Practically, this is a universal quantification of the expression to be checked without the need of more expressive language than CTL.
- Checking relation $\mathsf{pconf}_6^{v,v',K}$ requires additionally to check the number of rising edges on the Boolean outputs v and v'. We introduce a variable Q^v in the model $\Gamma_{M,M'}$ that is incremented by one if v has a rising edge at the end of a cycle and decremented by one if v' has a rising edge

Table 4.2: CTL representations of the pconf relations [*c*7]

M pconf M'	$\iff$	**Satisfaction of the CTL representation**
1.	$\mathsf{pconf}_1^{v,v'}$	$\Gamma_{M,M'} \vDash \mathsf{AG}(v = v')$
2.	$\mathsf{pconf}_2^{v,v',n}$	$\begin{cases} \Gamma_{M,M'} \vDash \mathsf{AG}(v = v'_{-n}) & \text{if } n \geq 0 \\ \Gamma_{M,M'} \vDash \mathsf{AG}(v_n = v') & \text{if } n < 0 \end{cases}$
3.	$\mathsf{pconf}_3^{v,v',\{k_1,k_2,\dots\}}$	$\bigvee_{i\in\{k_1,k_2,\dots\}} M \ \mathsf{pconf}_2^{v,i} \ M'$
4.	$\mathsf{pconf}_4^{v,v',K}$	$\Gamma_{M,M'} \vDash \mathsf{AG}($ $(\bigvee_{i\in K} v_{-\mu(K)} = v'_{i-\mu(K)}) \wedge$ $(\bigvee_{i\in K} v_{-i-\mu(-K)} = v'_{-\mu(-K)}))$ where $\mu(K) \triangleq \max(K \cup \{0\})$
5.	$\mathsf{pconf}_5^{v,v',K}$	$M \ \mathsf{pconf}_4^{v,v',K} \ M' \wedge \mathsf{AG}\ \mathsf{AF}(P^v = 0)$
6.	$\mathsf{pconf}_6^{v,v',K}$	$M \ \mathsf{pconf}_4^{v,v',K} \ M' \wedge \mathsf{AG}\ \mathsf{AF}(Q^v = 0)$

Table 4.3: Translation of CTL temporal operators (based on [*c*7])

Expression on PLC model *(1 transition = 1 cycle)*	**Expression on verification model (IM)** *(1 transition = 1 computation)*
$\mathsf{AF}(\alpha)$	$\mathsf{AF}(EoC \wedge \alpha)$
$\mathsf{AG}(\alpha)$	$\mathsf{AG}(EoC \rightarrow \alpha)$
$\mathsf{A}[\alpha\ \mathsf{U}\ \beta]$	$\mathsf{A}[(\alpha \vee \neg EoC)\ \mathsf{U}\ (EoC \wedge \beta)]$
$\mathsf{AX}(\alpha)$	$\mathsf{AX}\ \mathsf{A}[(\neg EoC)\ \mathsf{U}\ (EoC \wedge \alpha)]$

at the end of the cycle. Therefore $Q^v = 0$ means that v and v' had the same number of rising edges in the preceding cycles. The additional requirement is expressed as "Q^v has to go back to zero always in the future".

Verification model semantics. The CTL expressions in Table 4.2 were defined assuming that one transition in the verification model represents one PLC cycle, e.g. $\mathsf{AG}(v = v')$ means that v and v' equal to each other *at the end of all PLC cycles*. The semantics of the intermediate model is defined differently: one transition in $\Gamma_{M,M'}$ represents only one computation. However, the states representing ends of PLC cycles are labelled by the label EoC (end of cycle). Based on that the CTL expressions can be systematically transformed to have the same meaning on the verification model as before. These transformations can be seen in Table 4.3. The idea of the translation is general and can be extended to all LTL temporal operators as well.

Publications related to this section. The detailed discussion of conformance checking and the need for permissive conformance relations were presented in [*c*6]. The case study in [*c*9] used PLCspecif and conformance checking to verify a real-life safety logic.

4.7 Evaluation and Usage Examples

This section is dedicated to the evaluation of the proposed formal specification language and the connected methods. First, Section 4.7.1 compares the proposed specification language and the connected methods to the design requirements discussed in Section 4.1. After that, usage examples of the specification and analysis methods discussed in this chapter. The examples focus mainly on conformance checking, but they also implicitly demonstrate the feasibility and applicability of the proposed specification methods in the mentioned cases. The examples are from the development of the previously introduced UNICOS framework's reusable baseline objects and from the development of the SM18-PLCSE system. For the introduction of the use cases (UNICOS baseline objects and SM18-PLCSE) the reader is referred to Section 3.8.

It has to be noted that the implementation of PLCspecif is in a proof-of-concept phase at the moment, the production-ready implementation is a future work and not in the scope of this thesis. The described measurements were executed on a PC with the following configuration: Intel Core i7-3770 3.4 GHz CPU, 8 GB memory with Windows 7 x64 and .NET 4.0 framework. The underlying tools were the following: PLCverif 2.0.3, nuXmv 1.1.1 and Z3 4.3.0.

4.7.1 Comparison of PLCspecif and the Collected Requirements

This section overviews the requirements collected in Section 4.1 and shows how does PLCspecif respond to them.

GRa. **Lightweight method.** PLCspecif builds on the available domain knowledge by reusing formalisms that are familiar to the targeted users. Furthermore, it has a simplified semantics that needs only short training for new users.

GRb. **Domain-specific method.** PLCspecif targets the development of the PLC software modules from the beginning. The semantics is heavily adapted to the cyclic PLC programs. See later the details, at DRa.–DRe.

GRc. **Reduced expressivity.** To make the language less error-prone, the base formalisms are kept as simple as reasonably possible. For example, the state machine formalism does not allow the use of actions that could make the semantics complex and confusing. Besides, timing aspects are kept isolated, further simplifying the semantics. This also means that the certain programs cannot be described using state machine modules, but in these cases the input-output connection modules are typically more suitable.

GRd. **Variety of supported languages.** As previously mentioned, different formalisms may be suitable to describe different behaviours. This is why state machines, input-output connection diagrams and timer modules are all available to the user, who can choose the most appropriate one. Similarly, besides the common arithmetic format of the expressions, two tabular representations are supported which are more suitable for the description of complex expressions.

GRe. **Support for verification.** PLCspecif contains built-in support for verification. For example, the consistency and the well-formedness can be directly checked on the specification using static analysis. Invariant properties can be defined in the specification that can be verified directly on the formal specification, reusing the verification workflow presented in Chapter 3 and implemented in PLCverif. This is thanks to the semantics of PLCspecif which is designed in a way that an automaton representation of the specification can be easily generated. Furthermore, PLCspecif supports equivalence or conformance checking that is a powerful tool to compare for example an implementation and a specification, without the need of describing any requirements.

GRf. **Support for documentation.** PLCspecif supports free-text annotations which help the users to understand the specifications. The concrete syntax of PLCspecif is designed so that it can serve as a documentation too. The same annotations are present in the generated code, which also follows the structure of the specification.

DRa. **Events with proper semantics.** PLCspecif explicitly supports the definition of events. The events have priorities and in each (sub)module at most one event can trigger. This makes the execution deterministic and responds to the duality that PLC programs are not event-triggered (interrupt-driven), yet several input signals represent event-like operations. The strict precedence of events helps also to make the specification less error-prone and easier to understand.

DRb. **Clean core logic.** PLCspecif decouples the input handling, the output handling and the core logic description. This helps the specifier and the reader of the specification to focus on the key behaviour aspect, and then later on the precise definition of inputs and outputs.

DRc. **Hierarchical, modular structure.** PLCspecif is organised into modules which build up a hierarchy. This helps the specifier and the reader of the specification to focus on one aspect at a time first, then see greater and greater parts of the specification. The modularity also helps to reuse already existing submodules.

DRd. **Time-dependent behaviour.** As PLC programs often contain timing (using real time), special support is given for describing the time-dependent behaviour. However, handling time in state machines or data-flow representations may make the semantics complex and potentially confusing. Therefore time-dependent behaviour is extracted to isolated timer modules. These modules follow the standard semantics of the PLC timers defined in [I61131-3], thus they are familiar to the targeted users.

DRe. **Relaxed conformance relations.** Equivalence checking is an important verification tool to compare a specification and its implementation. However, in reality the strict equivalence checking reveals many differences that are acceptable in practice. To make it practically usable, more permissive conformance relations and their checking based on model checkers were defined, specifically targeting the domain of industrial control systems.

4.7.2 UNICOS Re-engineering[18]

To show the capabilities, the usage of PLCspecif and the related verification methods is described here. The library of baseline objects in UNICOS has been used for more than ten years. As the requirements often change, many extensions and modifications were made. In certain cases some constraints were applied to the modifications, thus currently the implementation is suboptimal. Furthermore, many specifications documentations are now out-of-date, causing discrepancies between the implementation and the informal documentations. This is partially because the behaviour of the objects became too complex, and it is difficult describe them in a specification. For example, see Figure 4.15(a) that is the specification of a state-based submodule of a baseline object. At a first glance it might look like a precise definition, but a closer look at the state machine and the implementation reveal that there are various discrepancies, missing information, unresolved conflicts, etc. Furthermore, the analysis of the source code can show that there is no strict precedence between the various signals. Although this mode manager submodule is used in all applications, interviews with the developers revealed that they are not confident about the behaviour of the object in special cases. Figure 4.15(b) shows the complete description of the same submodule in PLCspecif (only the core module description is shown,

[18]This part is an extension of the relevant part of Section V of [*c*6].

the thicker green edges represent the non-event-triggered transitions). This description is clean and unambiguous.

The verification workflow proposed in Chapter 3 was applied to check certain properties of these baseline modules. If a problem was found, its root cause was fixed. However, this only provides small functionality fixes, but not the rethinking of the overall expected behaviour. Therefore, to improve the quality of the implementation and the specification, the following approach was proposed. PLCspecif could be used to formally capture the intended behaviour of the baseline objects. After the specifications are developed, the conformance between the legacy code and the new formal specification can be performed. It is known that certain behaviours of the objects are non-intuitive or inconsistent, and we could benefit from this re-engineering campaign to reconcile these problematic parts. Therefore in certain cases strict equivalence will not be required. After the formal specifications are ready, the new implementations can be generated using the code generators.

Experiments for this project were started by capturing the formal specifications of certain baseline objects. It showed that it is feasible to develop PLCspecif specifications for the UNICOS baseline objects, also that conformance checking and code generation is feasible based on these models. However, during specification many problematic parts were identified, which require the detailed analysis of the domain experts. While this means that this experimental project will run for a long time, it has already been shown that constructing formal specifications – although this needs significantly more effort and time than writing informal specifications, even if the specification method is targeting specifically the application domain – may reveal deep, hidden design flaws.

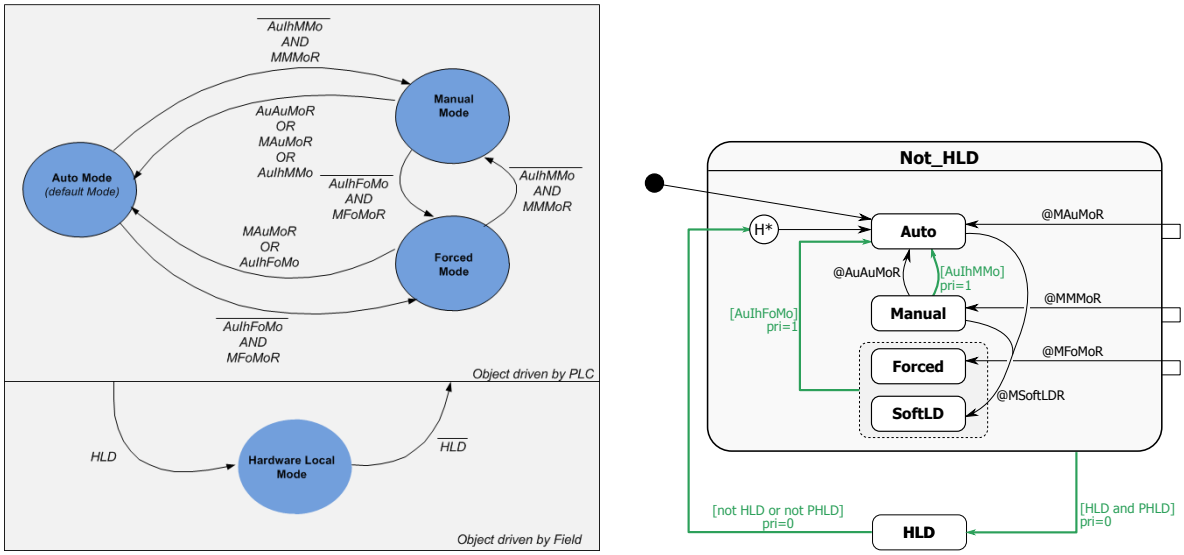
It has to be noted that certain submodules of these baseline objects describe complex behaviours, for example the one in Figure 4.15(b). The generated source code corresponds to the specified behaviour and follows the structure of the specification, but is not optimal and too long. This is due to the simple text-to-text transformation which does not check the satisfiability of the generated conditions. By eliminating the conditional statements with unsatisfiable conditions, the generated code could be made much more practical, but optimisation of the generated code is a future work after the AST-based code generation workflow will be implemented.

Usage of the permissive relations in practice. In the following a certain aspect of this work will be shown: the usage of permissive relations in practice (based on [*c*6]). In this case we used an object from the UNICOS framework's object library, widely used at CERN for PLC software. We introduced a small modification in the code causing a one-cycle-long delay in two of the outputs used for informational purposes, i.e. a delay that does not cause any problems. By using strict equivalence checking, the two code versions differed, a difference was found in less than a second[19].

Without permissive conformance relations the conformance checking would get stuck: having and analysing only a single counterexample, it is not known whether any other important differences exist, i.e. if the modification caused any relevant side-effect.

We repeated the conformance checking, but taking into account that various conformance relations can be used for the different outputs. First, we identified the level of required conformance between each of the corresponding output pairs. We selected a permissive conformance relation ($\mathsf{pconf}_4^{v,v',\{-1,0,1\}}$) for the two outputs where the strict equivalence is not required. After this, the conformance checking demonstrated in less than a second that the two code versions are conformant, there is no inequivalence besides the permissible delay of the above-mentioned variables. This example shows the advantage of the permissive conformance relations: using them the developer can

[19]This validation example is published, refer to DOI 10.5281/zenodo.45415.



(a) Current description (source: http://cern.ch/ucpc-resources/1.8.0/objects/on-off.html)

(b) Proposed description in PLCspecif

Figure 4.15: Representations of the mode manager submodule of the field objects

be sure that his modifications did not affect any other behaviours, and also it did not cause intolerable changes in the "informative" outputs.

The reachable state space of each PLC program model was around 10^8 states (before reductions), while the composite model contained only 7×10^4 reachable states (after reductions). The execution time of the nuXmv model checker was 0.6 s for the conformance checking using both strict and permissive relations.

4.7.3 SM18-PLCSE Safety Controller

To show the applicability of the PLCspecif specifications and the connected methods, we refer back to the SM18-PLCSE safety controller, already introduced in Section 3.8.2. This case also shows that the proposed conformance checking workflow may scale up to models with large state spaces.

When the development of the new SM18-PLCSE safety controller started, the PLCspecif specification and the connected analysis methods were not ready yet to be used. Therefore at the beginning of the development we used only requirement patterns for formal requirements specification, and PLCverif to verify these requirements. As discussed in Section 3.8.2, the requirement pattern-based description of the requirements was a complex task and it demonstrated that the detailed analysis of such systems needs different methods.

Conformance checking. Later, when PLCspecif became ready, the formal specification of the safety logic of SM18-PLCSE was developed based on the semi-formal requirements given by the client. This safety logic has a simple structure, however it uses many input and output variables. The behaviour in PLCspecif was mostly described using the tabular expression representations. This resulted in a specification similar to the original semi-formal requirement table, but with precise, formal semantics and verification possibilities. Then conformance checking between the specification and the implementation was performed. It has to be noted that as this is a stateless safety logic, there was no

need to use the permissive conformance relations, instead strict equivalence checking was applied. Using PLCverif and requirement patterns we found 12 issues which were fixed in the implementation. We gained confidence in the correctness of the implementation by extensively using model checking. The conformance checking was performed only to try the novel methods.

Although the implementation was already checked using model checking, by comparing the specification and the implementation, two additional discrepancies were found. These problems were not covered by the requirements before, underlining that providing a complete behaviour description using temporal logic or requirement patterns only is extremely difficult, even for users experienced in formal verification. After fixing these two problems, the conformance of the specification and the implementation was proven. The composite IM generated for this case was more complex than the IMs generated for the verification of single requirements. Therefore the verification run time was longer as well, proving the conformance (evaluating the CTL expression representing the selected conformance relations on the composite model) took 482 s [*c*6]. However, this verification had to be done only once, while the requirement pattern-based approach required 24 separate runs for the 24 different requirements. Therefore although the conformance checking took more time in this case than verifying a single requirement, the overall verification was faster using conformance checking.

It has to be noted that as conformance checking typically results in more complex models with larger state spaces, in certain cases the conformance checking will not be feasible, while verifying individual requirements might be still possible. Furthermore, developing formal specification might need significant effort compared to defining certain critical requirements, but may imply a higher level of confidence in the correctness of the developed software. This shows that requirement pattern-based model checking and formal specification-based conformance checking are complementary methods with different strengths and weaknesses.

The SM18-PLCSE safety controller is in use since August 2015. To this day no issues were found related to the implementation of the safety logic.

Code generation. In case of safety-critical PLCs provided by Siemens the programs have to be manually developed in graphical languages, both code generation and the usage of the SCL language are restricted. However, in order to check the code generation workflow and the different use cases of conformance checking, an SCL implementation of the safety logic was generated from the PLCspecif specification. Then conformance checking was performed between the real (STL) and the generated (SCL) implementation. The conformance was proven in this scenario too, and the execution time was nearly identical to the specification to implementation comparison.

The generated code is only 279 lines long, and most of it consists of variable and constant definitions. The implementation itself mainly consists of some complex but consistent variable assignments, following the structure of the specification. Unfortunately, the generated code cannot be used in practice, as safety Siemens PLCs have to be programmed manually in FBD or LAD.

Publications related to this section. Conformance checking-related uses were reported in [*c*6]. Different aspects of the SM18-PLCSE conformance checking case study were presented in [*c*6; *c*9].

4.8 Summary and Future Work

This chapter describes PLCspecif, a formal behaviour specification language that has been specifically designed for the development needs of PLC software modules. Besides the unambiguous, domain-specific behaviour description, PLCspecif allows the usage of various methods:

- **Static analysis.** To check the well-formedness of the specification, I defined a set of well-formedness rules. Most of the checks for the rules are implemented using simple graph algorithms. For some others I provided translations into SAT problems.
- **Verifying invariant properties.** When a PLC developer cannot be sure that the imperative behaviour description satisfies some desired invariant properties, it is possible to include these properties explicitly in the specification. Based on the semantics definition of PLCspecif, I gave a translation to PLCverif's intermediate model formalism. The previously discussed pattern-based requirement specification is used for the definition of invariant properties.
- **Code generation.** As a PLCspecif specification describes the complete behaviour of a PLC program, it is possible to generate automatically an equivalent implementation. I defined the formal semantics of PLCspecif in a way that it is close to a control flow graph (or control flow automaton). Based on this, I designed and developed a code generation method that builds Siemens SCL code for a given PLCspecif specification. For maintainability reasons the generated code is designed to be readable and understandable, following the structure of the specification.
- **Conformance checking.** PLCspecif supports the conformance checking between the specification and implementation. For this, I reused the previously described method to translate the specification into intermediate models used by PLCverif and designed a way create a composite model of the two artefacts. With this composite model a model checker via PLCverif can show or refute the equivalence.

  In practice, PLCs are often used to control slow processes, where small delays in the output do not have any observable effect on the controlled process. Requiring a strict equivalence between two artefacts in these cases results in false positives, i.e. differences that are acceptable for the users. I identified typical cases when strict equivalence is not required between a pair of outputs in the checked artefacts, and I defined more permissive conformance relations. This way the PLC developers can define for each output pair the level of conformance they require.

The contributions targeted in this chapter were the following.

Thesis 3 I designed *PLCspecif*, a formal behaviour specification language adapted to the needs identified in PLC program development. This language is designed to be used for code generation and verification purposes.

3.1 I designed the main language concepts, then I defined the precise syntax and semantics of this language. The design of the language is based on collecting the requirements by analysing the literature of formal specification methods and on the feedback from the PLC developer community at CERN.

3.2 I developed a transformation from the PLCspecif specification to the intermediate model (IM) language used by PLCverif. This allows the usage of various model checkers to verify the invariant properties of the specification.

3.3 I designed and implemented an automated code generation method for the PLCspecif specification language based on its formal semantics. This code generation method is flexible and configurable, and produces Siemens SCL code that is systematically derived from the formal specification.

3.4 I designed new conformance relations, which can be used in the PLC software development domain and allow designers to define acceptable discrepancies between the specified

> and implemented behaviours (such as short delays in output signals). I defined a conformance checking method based on model checking to determine whether or not a relation holds for an implementation-specification pair. Accordingly, these relations provide means to verify a legacy implementation or a manually-modified generated code against a specification. I provided an implementation for checking these relations and evaluated their practical applicability.

The design requirements, and the syntax and semantics of PLCspecif (Thesis 3.1) were discussed in Sections 4.1 and 4.3. The verification of invariant properties (Thesis 3.2) using model checkers was discussed in Section 4.4. The code generation method (Thesis 3.3) that can automatically provide implementation for a given PLCspecif specification was presented in Section 4.5. The design and implementation of various conformance relations and the corresponding conformance checking methods (Thesis 3.4) were presented in Section 4.6.

Future work. PLCspecif is currently in a proof-of-concept implementation phase. This allowed to evaluate certain aspects of the language and the attached methods, but more work is needed in order to make it usable in the everyday development, directly by the industrial practitioners. The main future works are the following.

- A production-ready implementation is needed that can be given directly to the users. After this, the method can be validated with PLC developers who are not experienced in formal methods.
- PLCspecif could be used with additional methods too, for example [Uni15] provided an initial approach to generate test cases based on this specification language.
- The performance of the conformance relations should be improved. While it was possible to check the conformance between the specification and the implementation in case of the large SM18-PLCSE implementation, smaller objects with complex behaviours cause too long run times in some cases.
- To make the code generator production-ready, the simple text-to-text generation should be replaced by a generation through the AST of the target language. This would allow to simplify the code before generation, e.g. removing the conditional statements with unsatisfiable conditions.
- In the future, PLCspecif should be included in a broader specification and verification methodology that provides solution for the specification of complete PLC control applications, not only for PLC modules.

Chapter 5

Summary of the Research Results

This final chapter reviews the presented new contributions of this work. The challenges discussed in Section 1.1.3 will be revisited together with the given solutions in Section 5.1. Section 5.2 summarises the various applications of model checking in this work. Finally, Section 5.3 concludes the chapter with the theses of this dissertation.

This dissertation proposed various improvements for formal methods to make them more applicable in the development of PLC-based industrial control software, targeting both the performance and usability issues.

Performance improvements. Chapter 2 presents an extension of the saturation-based model checking techniques, which resulted in B-I-Sat, a novel bounded saturation-based CTL model checking algorithm. This verification method is based on direct modelling, thus targets scenarios when the development of a dedicated workflow is not feasible or not efficient. This novel algorithm improved the performance of CTL model checking in many cases compared to the original, non-bounded saturation-based CTL model checking.

In some cases the usability and performance improvements overlap: if the verification takes excessive amount of time, the method is not usable in practice. The workflow introduced in Chapter 3 mainly targets the improvement of usability. However, for this reason automated reductions were included, improving the performance of the workflow.

Usability improvements. Chapters 3 and 4 provided formal verification and specification methods specifically designed for and adapted to the needs of industrial control software development. In these cases direct modelling or usage of verification tools is not needed, thus these are suitable solutions for developers without extensive formal methods knowledge. Different methods with different use cases were proposed which are summarised in Table 5.1.

The verification workflow presented in Chapter 3 and implemented in the PLCverif tool targets case 1 of Table 5.1. It is applicable when no formal specification is available and it can be used based on the available implementation and some informal specification, which is adapted to the requirement pattern-based inputs of the workflow by the user.

The second and third use cases consider situations when a formal specification is available. In Chapter 4 PLCspecif, a novel formal specification language was introduced that specifically targets the complete behaviour description of PLC modules or logics. Conformance checking (case 2 in Table 5.1) provides a way to formally check the correspondence between the specification and a manually

developed implementation. This may be the appropriate solution for the cases with legacy applications or when manual implementation is mandatory. Case 3 of Table 5.1 targets projects where the development is started by creating a formal specification. This specification can be checked using static analysis techniques, then a code generator may automatically produce an implementation with equivalent behaviour.

By supporting various use cases it is possible to find the most appropriate solution for the different development situations and requirements. This variety of proposed solutions also helps the step-by-step introduction of formal methods to the development processes [*c*8].

Table 5.1: Use cases for application of formal methods without direct modelling [*c*8]

	Available			
	Informal specification	**Manual implementation**	**Formal specification**	**Action**
(1)	+	+		Model checking of the manual implementation
(2)	+	+	+	Conformance checking between manual implementation and formal specification
(3)	+		+	Code generation based on checked formal specification

5.1 Responses to the Challenges

In the following the various responses given to the initial challenges are summarised.

Challenge 1: Designing model checking algorithms combining bounded and saturation-based techniques to improve their performance. *Both bounded model checking and saturation-based techniques increase the set of models on which verification is feasible compared to basic explicit model checking algorithms. Is it possible to combine these two approaches? Does it improve the performance with respect to the original saturation-based model checking?*

The work presented in Chapter 2 introduces B-I-Sat, a saturation-based bounded CTL model checking algorithm. It shows the feasibility of designing an efficient algorithm combining bounded model checking techniques with saturation-based techniques. The measurements in Section 2.6 demonstrate that B-I-Sat provides lower execution times and better scalability in certain cases, when the given formal requirement can be evaluated by exploring only a part of the model.

Challenge 2: Making model checking easily accessible to the PLC developers. *Model checking is rarely used for industrial control software, mainly because of the enormous effort needed to create formal models and requirements, furthermore to learn the usage of the model checker tools. How can model checking be made accessible and practically applicable in the PLC program development process? How can model checking be used without excessive effort, without exposing the users (PLC developers) to formal details, to complex mathematical formalisms?*

The response to this challenge was the design of a verification workflow introduced in Sections 3.2 and 3.4 that hides the complex, formal details of model checking. The basic design of the workflow was a joint work with B. Fernández. Chapter 3 discusses the own contributions: the detailed design of the workflow and the development of PLCverif, a tool which makes model

checking accessible to the developers without extensive training. To be able to do this, automated transformations and intermediate model formalisms were developed. The result is a verification workflow that relies on the implementation and requirements given by filling requirement patterns, and that produces the results as verification reports in an easy-to-understand format.

Challenge 3: Making the PLC model checking applicable to real-world PLC programs. *The formal models of real PLC modules or programs and their state spaces tend to be extremely large, making the model checking infeasible using general-purpose model checker tools. Could heuristic model reductions reduce the performance needs of model checking and therefore cope with a bigger set of models?*

In order to make the verification workflow described in Chapter 3 practically feasible, reduction heuristics were designed and developed based on the intermediate model language that is a model checker-independent behaviour description of the software to be verified (see Section 3.5). This way every supported model checker may benefit of the reductions. The evaluation in Sections 3.5 and 3.8 demonstrates that the reductions can make the verification of real-life PLC programs feasible, which could not be done without the reductions in the presented cases.

Challenge 4: Extending the model checking approach to safety-critical PLC programs. *The original PLC model checking workflow supported the Siemens SCL language only, which – being a high-level language – is more suitable for the implementation of complex programs. However, the development of PLCs used in safety-critical settings has specific procedures and restrictions, such as the mandatory usage of FBD or LAD languages (in case of Siemens PLCs). How can model checking be adapted to these lower-level programming languages used in safety-critical PLC program development?*

The verification workflow and the implementation initially targeted the analysis of PLC programs written in SCL language. To support the verification of safety-critical PLCs, support for the low-level STL language was developed. The verification of FBD and LAD programs is also possible through this language. For this reason, the precise semantics of STL was analysed and explored, a partial translation between STL and SCL languages was developed that targets the verification of simple safety programs, and the set of reduction heuristics was extended (see Section 3.6). These works allowed to apply the proposed verification workflow for example to the SM18-PLCSE safety logic, a safety-critical controller developed and used at CERN. The formal verification of SM18-PLCSE successfully revealed various problems before putting the system into production.

Challenge 5: Providing lightweight formal specification for PLC software modules. *Unambiguous requirements are essential for any development or verification activity. Formal specifications may reduce the ambiguity, but the general-purpose formal specification methods are too complex and non-intuitive to be used in the PLC development domain with a reasonable effort. What are the requirements towards a formal specification language specially adapted to the PLC domain? What formal specification method can aid the PLC program development process?*

Sections 4.1 and 4.2 provide an overview of the available formal specification methods and the requirements towards a method that can be applicable to the industrial control software development. Based on this overview a new formal specification language (PLCspecif) was designed, which is described in Section 4.3. This complete behaviour specification language is suitable for the development and verification of the targeted software. As PLCspecif is designed

specifically to describe PLC modules, its usage does not require deep formal methods expertise.

Challenge 6: Providing verification solutions based on formal specification. *How could formal specification improve the PLC program verification? What verification methods can be used to check the conformance between a PLC program and its formal specification? How can this be made useful in practice, without excessive amount of false positives (i.e. without having a high number of detected differences that are considered to be acceptable by the developers)?*

The PLCspecif formal specification language is incorporated with a variety of additional verification and development methods: code generation, static analysis, and conformance checking. Conformance checking allows the precise comparison of the two artefacts' behaviours, i.e. the behaviour of a given specification and implementation. This is especially important when code generation is not applicable. In industrial control software it is often observed that the strict equivalence between the specification and the implementation is not required, as the controlled process might be orders of magnitudes slower than the controller, thus small delays might be acceptable. Therefore various, more permissive conformance relations and methods to evaluate their satisfaction were designed and presented in Section 4.6.

5.2 Summary of the Proposed Verification Methods

The different contributions propose the usage of model checking in various settings, e.g. using different models and requirements. Figure 5.1 and Table 5.2 compare the main aspects of the proposed approaches for various use cases.

1. **Direct model checking** (Thesis 1). In the first thesis, model checking is used without any additional aid or adaptation. The formal models and requirements are created manually, in this case using Petri nets and CTL. Then the B-I-Sat model checking algorithm (implemented in the PetriDotNet framework [*c*10]) is executed and its results are evaluated manually. This is a suitable method, for example, when a non-deterministic model is needed that over-approximates the set of behaviours, or when the verification is a rare event and it is not efficient to develop a dedicated toolchain for the given platform, e.g. in case of the verification of the so-called PRISE safety logic. This approach is illustrated in Figure 5.1(a).

2. **Model checking based on requirement patterns and PLC code** (Thesis 2). The inputs of this workflow (implemented in PLCverif [*c*13]) are the implementation (PLC code) and a filled requirement pattern. Both are understandable for the PLC developers. The implementation is then automatically translated to an intermediate model (IM). A wrapper of the model checker takes care of transforming the model and requirement from an intermediate representation to the concrete syntax of the model checker. The result of the verification process is a verification report, readable for the users. This workflow is suitable for users without extensive formal verification knowledge. Additionally, thanks to the automation and adaptation, this method can be included in the PLC development process efficiently, without excessive effort. This approach is illustrated in Figure 5.1(b).

3. **Conformance checking** (Thesis 3). Conformance checking takes a formal specification besides the PLC code as input. Both artefacts can be transformed into IM, then reduced and combined into a composite model in order to perform conformance checking. Based on the selected conformance relations a temporal logic requirement is generated. Then the wrapped model

checker evaluates the satisfaction of this requirement on the composite model, similarly to use case #2. The result is presented in a verification report. If a formal specification is available, conformance checking provides an easy-to-use and thorough way to compare the implemented and the expected behaviour. This approach is illustrated in Figure 5.1(c).

4. **Code generation and invariant property checking** (Thesis 3). If the implementation is generated using a correct code generator, there is no need to compare the behaviours of the implementation and the specification. However, in this case model checking can be used in the specification phase to ensure the satisfaction of invariant and/or safety properties. For this, the formal specification is transformed into IM, like in use case #3. The invariant properties are defined using requirement patterns. Then the wrapped model checker evaluates the satisfaction of the given invariant properties and a report is generated. This approach is illustrated in Figure 5.1(d).

Table 5.2: Overview of the proposed model checking methods

	#1 Direct model checking	**#2** PLC code model checking	**#3** Conformance checking	**#4** Invariant checking
Model	Direct modelling	IM from implementation Reduction	IM from implementation IM from formal specif. Composition Reduction	IM from formal specif. Reduction
Requirement	Manual	From requirement pattern	From conformance relation	From requirement pattern
Model checker	Unwrapped	Wrapped	Wrapped	Wrapped
Result	Raw	Verification report	Verification report	Verification report

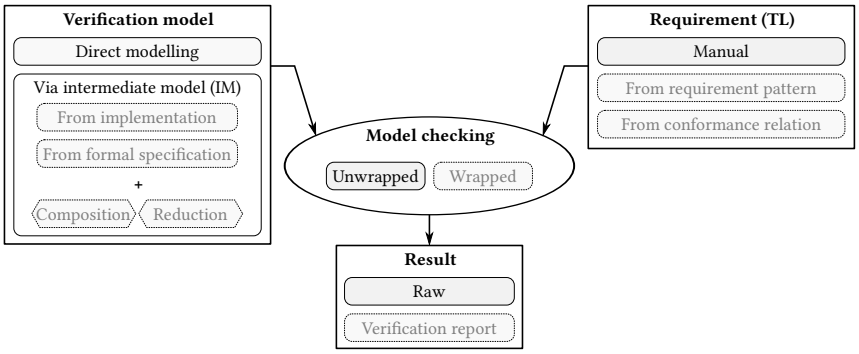
5.3 Summary of the Theses

This section overviews the theses and the corresponding own publications.

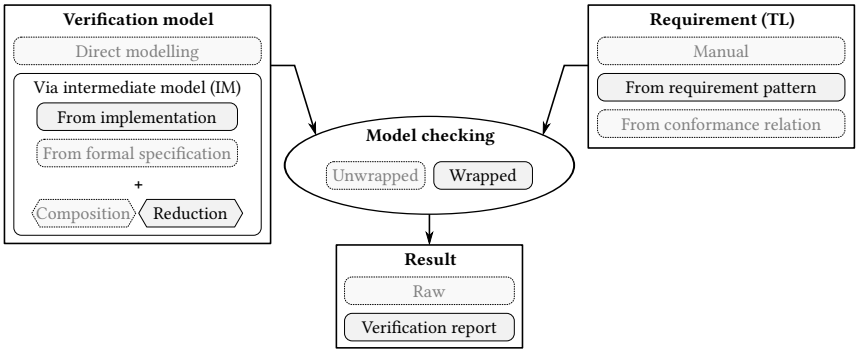
Thesis 1 I designed *B-I-Sat* (Bounded Iterative Saturation), a novel computation tree logic (CTL) model checking algorithm, that efficiently combines bounded model checking with saturation-based techniques.

- 1.1 I defined the building blocks, and based on them the B-I-Sat algorithm to perform bounded CTL model checking using saturation-based techniques. I defined two strategies for B-I-Sat: the restarting and continuing strategies.
- 1.2 I defined termination conditions for the B-I-Sat algorithm using three-valued logic.
- 1.3 I developed an advanced incremental search strategy, the so-called compacting strategy to reduce the memory consumption of the B-I-Sat algorithm.
- 1.4 I evaluated the performance of the B-I-Sat algorithm with the different strategies on various benchmark models and an industrial example.

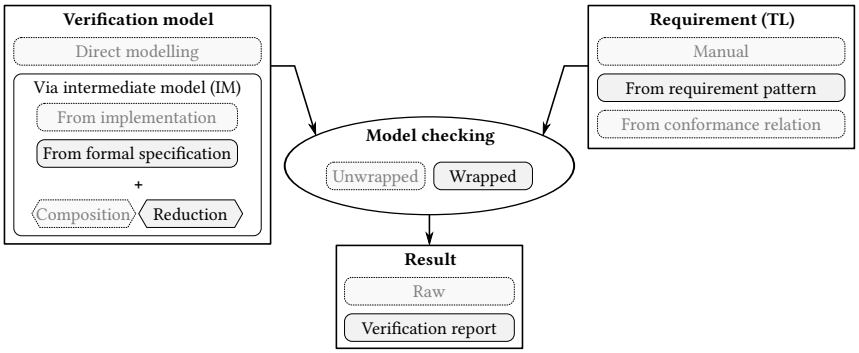
The results of Thesis 1 are presented in Chapter 2 of the dissertation. Related publications are the following: [*j*2; *j*4; *c*10; *c*17; *c*18; *e*20; *e*21].



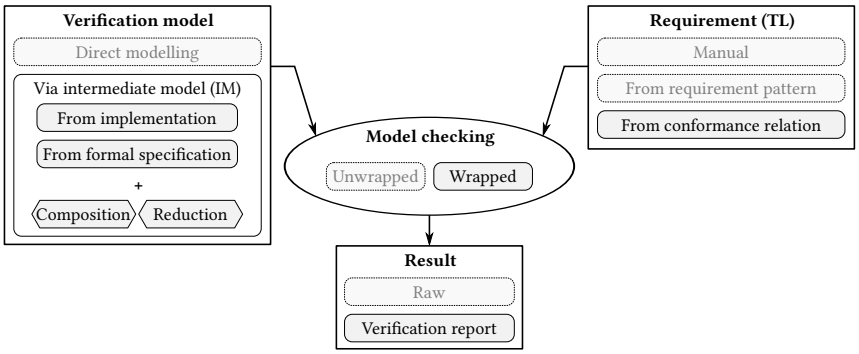
(a) B-I-Sat model checking method (Thesis 1)



(b) Model checking PLC programs (Thesis 2)



(c) Checking invariant properties (Thesis 3)



(d) Conformance checking (Thesis 3)

Figure 5.1: Overview of the proposed model checking methods

Thesis 2 I contributed to the development of a generic, flexible workflow to apply model checking to PLC programs without requiring extensive formal methods knowledge from the users. I designed essential parts of this workflow, as follows.

- 2.1 I designed an intermediate model (IM) language that can represent PLC programs and can act as a pivot language for different model checkers. The IM-based model checking is a fully automated method that can be used by developers who are not familiar with formal verification techniques.
- 2.2 I developed heuristics to automatically reduce the size of the intermediate models, making the model checking workflow less resource-demanding.
- 2.3 I extended this model checking workflow (originally developed only for SCL programming language) to support the PLC programming languages used in the development of safety PLCs: FBD and LAD, via the STL language.
- 2.4 I implemented the proposed model checking workflow in the *PLCverif* tool, providing push-button verification to the developers based on the source code of the PLC program and the pattern-based requirement specification. I evaluated the real-life applicability of this workflow using various PLC modules and applications developed and used at CERN.

The results of Thesis 2 are presented in Chapter 3 of the dissertation. Related publications are the following: [*j*1; *j*3; *c*8; *c*9; *c*11; *c*13; *c*14; *c*15; *c*16; *r*23; *r*24].

Thesis 3 I designed *PLCspecif*, a formal behaviour specification language adapted to the needs identified in PLC program development. This language is designed to be used for code generation and verification purposes.

- 3.1 I designed the main language concepts, then I defined the precise syntax and semantics of this language. The design of the language is based on collecting the requirements by analysing the literature of formal specification methods and on the feedback from the PLC developer community at CERN.
- 3.2 I developed a transformation from the PLCspecif specification to the intermediate model (IM) language used by PLCverif. This allows the usage of various model checkers to verify the invariant properties of the specification.
- 3.3 I designed and implemented an automated code generation method for the PLCspecif specification language based on its formal semantics. This code generation method is flexible and configurable, and produces Siemens SCL code that is systematically derived from the formal specification.
- 3.4 I designed new conformance relations, which can be used in the PLC software development domain and allow designers to define acceptable discrepancies between the specified and implemented behaviours (such as short delays in output signals). I defined a conformance checking method based on model checking to determine whether or not a relation holds for an implementation-specification pair. Accordingly, these relations provide means to verify a legacy implementation or a manually-modified generated code against a specification. I provided an implementation for checking these relations and evaluated their practical applicability.

The results of Thesis 3 are presented in Chapter 4 of the dissertation. Related publications are the following: [*c*5; *c*6; *c*7; *c*8; *c*9; *c*12; *e*19; *r*22].

Appendix A

Precise Definitions for the B-I-Sat Algorithm

In the following, the preliminaries intuitively introduced for Thesis 1 in Chapter 2 are precisely defined. This section is reusing the author's earlier work [*a*30].

Definition A.1 (Model checking problem [Cla08]). Let M be a Kripke structure (i.e. state-transition graph). Let f be a formula of temporal logic (i.e. the specification or property to be checked). Find all states s of M such that $M, s \vDash f$.

The goal of model checking is to find the $\{s \in \mathcal{S} \colon M, s \vDash f\}$ set (where $\mathcal{S}$ is the set of possible states in M, and the meaning of $M, s \vDash f$ is "the behaviour of the model M starting from state s (as an initial state) satisfies the property f").

Model checking is often considered as a decision problem, when the goal is to decide whether f is true for the initial state s_0 of the model M or not [CGP99][*a*30]. In these cases we call a requirement f satisfied iff $s_0 \in \{s \in \mathcal{S} \colon M, s \vDash f\}$ (or more precisely if $\mathcal{S}_0 \subseteq \{s \in \mathcal{S} \colon M, s \vDash f\}$, if multiple initial states $\mathcal{S}_0$ exist). Otherwise, the requirement is not satisfied. ▪

Definition A.2 (Kripke structure [CGP99]). A *Kripke structure* is a labelled directed graph, where the labelled nodes represent states, the edges represent possible state transitions. Given a set of propositions AP, it is a 4-tuple $M = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, L \rangle$, where:

- $\mathcal{S} = \{\mathbf{s}_1, \ldots, \mathbf{s}_n\}$ is a finite set of states;
- $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial states;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of possible state transitions; and
- $L \colon \mathcal{S} \to 2^{AP}$ is the labelling function. ▪

Definition A.3 (Distance of a state). Given a Kripke structure $M = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, L \rangle$, the *distance of a state* $\mathbf{s} \in \mathcal{S}$ is $\delta(\mathbf{s})$:

$$\delta(\mathbf{s}) = \begin{cases} 0 & \text{if } \mathbf{s} \in \mathcal{S}_0, \\ \min\{\delta(\mathbf{s}') : (\mathbf{s}', \mathbf{s}) \in \mathcal{N}\} + 1 & \text{otherwise.} \end{cases}$$ ▪

Definition A.4 (Petri net). A *Petri net* is a 5-tuple $PN = \langle P, T, E, w, M_0 \rangle$, where:

- P is a finite set of places;
- T is a finite set of transitions ($P \cap T = \emptyset$ and $P \cup T \neq \emptyset$);
- $E \subseteq (P \times T) \cup (T \times P)$ is a set of arcs;
- $w \colon E \to \mathbb{Z}^+$ is a weight function; and
- $M_0 \colon P \to \mathbb{N}$ is the initial marking (a *marking* is a function assigning token numbers to places) [Mur89].

A transition t is *enabled*, if each input place p_{in} of t is marked with at least $w(p_{in}, t)$ tokens [Mur89]. If an enabled transition t *fires*, it removes $w(p_{in}, t)$ tokens from each input place p_{in} of t and it adds $w(t, p_{out})$ tokens to each output place p_{out} of t, resulting in a new marking. ▪

Definition A.5 (Computation tree). Let M be a model with the set of states $\mathcal{S}$ and let $\mathcal{N}$ be the set of possible state-state transitions in the model. A *state path* is a (finite or infinite) sequence of states $(s_0, s_1, \dots)$ such that $\forall i : (s_i, s_{i+1}) \in \mathcal{N}$. For any state $s_0 \in \mathcal{S}$, a *computation tree* is a tree prefix rooted at s_0 containing every state path.

Note that computation trees are acyclic, therefore multiple nodes can represent the same state. Moreover, the computation tree is often infinite. ▪

Definition A.6 (CTL formula [CGP99]). A CTL formula (*state formula*) can be defined by the following rules:

- Every P atomic proposition is a state formula.
- If p and q are state formulae, then $\neg p$, $p \vee q$, and $p \wedge q$ are state formulae too.
- If p and q are state formulae, then $\mathsf{X}\ p$, $\mathsf{F}\ p$, $\mathsf{G}\ p$, and $p\ \mathsf{U}\ q$ are path formulae.
- If s is path formula, then $\mathsf{E}\ s$ and $\mathsf{A}\ s$ are state formulae.

The given definition permits eight possible temporal operator pairs: EX, EF, EU, EG, AX, AF, AU, AG. These are the building blocks of the CTL formulae. The intuitive semantics of these operator pairs are the following:

- EF p: p is true for at least one state on some (at least one) path;
- EG p: p is true for all states on some path;
- EX p: p is true for at least one of the next states;
- E$[p\ \mathsf{U}\ q]$: p is true for a state on some path and for all intermediate states on those path q is true;
- AF p: p is true for at least one state on all paths;
- AG p: p is true for all states on all paths;
- AX p: p is true for all next states;
- A$[p\ \mathsf{U}\ q]$: p is true for a state on all paths and for all intermediate states in all paths q is true.

Notice that a state formula with a temporal operator pair is also a state formula, thus "nesting" temporal operators is allowed. For example, the following is a valid CTL formula: AG $(q \vee \mathsf{AF}\ p)$. ▪

Definition A.7 (Bounded model checking (generalised)). Given a Kripke structure $M = \langle \mathcal{S}, \mathcal{S}_0, \mathcal{N}, L \rangle$, the *submodel within distance* b is a structure $M_{[0;b]} = \langle \mathcal{S}_{[0;b]}, \mathcal{S}_0, \mathcal{N}_{[0;b]}, L \rangle$, where:

- $b \in \mathbb{N}$;

- $\mathcal{S}_{[0;b]} \triangleq \{\mathbf{s}\colon \mathbf{s} \in \mathcal{S}, \delta(\mathbf{s}) \leq b\}$, i.e. the states in $\mathcal{S}$ within distance b;
- $\mathcal{N}_{[0;b]} \triangleq \{(\mathbf{s}, \mathbf{s}')\colon (\mathbf{s}, \mathbf{s}') \in \mathcal{N}, \mathbf{s} \in \mathcal{S}_{[0;b]}, \mathbf{s}' \in \mathcal{S}_{[0;b]}\}$, i.e. the state transitions between states in $\mathcal{S}_{[0;b]}$.

Then the generalised bounded model checking problem is determining whether $M, s \vDash f$ holds, based on models $M_{[0;0]}, M_{[0;1]}, \ldots$. ▪

Definition A.8 (Multivalued decision diagram [Cia07][*j25*]). A *multivalued decision diagram (MDD)* encodes a function $f\colon (\bigtimes_{i=K,\ldots,1} D_i) \to \{0,1\}$, where each D_i is a finite *domain* $D_i = \{0, 1, \ldots, n_i\}$. This is achieved by a structure $MDD = \langle V, level, r, children, value\rangle$, where:

- V is a finite set of *nodes*;
- $level\colon V \to 0, 1, \ldots, K$ is a function assigning *level numbers* to each node ($V_i \triangleq \{v \in V\colon level(v) = i\}$);
- $r \in V$ is the root node ($v \in V : level(v) = K \Rightarrow v = r$);
- $children\colon V_i \times D_i \to V_{<i}$ is a function defining edges between nodes, labelled by items of D_i ($V_{<i} = \bigcup_{j=0}^{i-1} V_j$; $children(v, d) = w$ it is often denoted by $v[d] = w$);
- $value\colon V_0 \to \{0, 1\}$ is a function assigning a binary value to each *terminal* node (i.e. nodes at level 0). ▪

Definition A.9 (Edge-valued decision diagram [CS02][*a30*]). An *edge-valued decision diagram (EDD)* encodes a function $f\colon (\bigtimes_{i=K,\ldots,1} D_i) \to (\mathbb{N} \cup \{\infty\})$, where each D_i is a finite *domain* $D_i = \{0, 1, \ldots, n_i\}$.

This is achieved by a structure $EDD = \langle V, level, r, children, value, \rho\rangle$, where:

- $V = V' \cup \{\bot\}$ is a finite set of *nodes*, where $\bot$ is the only terminal node (leaf), every other $v \in V'$ nodes are non-terminal nodes ($\bot \notin V'; V_i \triangleq \{v \in V\colon level(v) = i\}$);
- $level\colon V \to \{0, 1, \ldots, K\}$ is a function assigning *level numbers* to each node ($level(\bot) = 0, \forall v \in V' : level(\bot) > 0$);
- $r \in V$ is the only root node ($v \in V : level(v) = K \Rightarrow v = r$);
- $children\colon D_i \times V \to V_{<i} \times \mathbb{N} \cup \{\infty\}$ is a function defining edges between nodes, labelled by items of D_i ($V_{<i} = \bigcup_{j=0}^{i-1} V_j$) and non-negative integers or infinity ($children(d, v) = (s, w)$ is often denoted by $v[d].node = w, v[d].label = s$ or $v[d] = \langle s, w\rangle$);
- The root node r has a "virtual" incoming edge with a label ρ assigned to it. This ρ value is the so-called *dangling edge weight*. ▪

See Def. A.10 and Def. A.11 for special EDDs and Def. A.12 for the semantics of EDDs.

Definition A.10 (Canonical EDD [CS02][*a30*]). An EDD is *canonical* if for every $v \in V'$ there is at least one outgoing edge labelled with zero [CS02]. ▪

Definition A.11 (Quasi-reduced EDD [CS02][*a30*]). An EDD is *quasi-reduced* if *(i)* it is canonical, *(ii)* there are no duplicates on a level (i.e. if for $v, w \in V$ $level(v) = level(w)$ and $\forall i \in D_{level(v)}$: $v[i] = w[i]$, then $v = w$.), and *(iii)* for every $v \in V'$ every outgoing edge points to a node on level $level(v) - 1$ [CS02]. ▪

Definition A.12 (Semantics of an EDD [*a*30]). For simplicity, let us assume $\bot[i] = \langle \infty, \bot \rangle$ for any i. Informally, the r-rooted quasi-reduced EDD (with a dangling edge labelled with ρ) represents the following $f(x_n, \ldots, x_1)$ function:

$$\begin{aligned} f(i_n, \ldots, i_1) = m \quad \Leftrightarrow \quad & r[i_n] = \langle m_{n-1}, v_{n-1} \rangle, \\ & v_{n-1}[i_{n-1}] = \langle m_{n-2}, v_{n-2} \rangle, \\ & \ldots, \\ & v_1[i_1] = \langle m_1, \bot \rangle, \\ & m = m_1 + \ldots + m_{n-1} + \rho \end{aligned}$$

▪

Appendix **B**

Pseudocode of the Bounded Saturation Algorithms

The following part describes the proposed B-I-Sat algorithm and the different strategies in more detail. The pseudocode is based on the author's earlier work [*a*30]. The pseudocode uses the structure *BoundedSaturationData*, presented in Figure B.1. This represents the interface of the basic EDD-based bounded state space exploration algorithm, extended by the constrained and negated constrained saturations' extra checks.

By invoking the *BoundedSaturation()* method of *BoundedSaturationData*, the state space exploration with the previously set parameters will be executed. Before the execution, the following fields have to be set: `bound` that determines the current bound value, `initial` that is the root edge of the initial state space (encoded as EDD), the `pruningMethod` that determines the applied pruning method (exact or approximate). Optionally, two constraints can also be defined using their MDD's root node: `constraint` for constrained saturation and `negConstraint` for the negated constrained saturation. After execution, the `stateSpace` field will determine the root edge of the state space EDD, and `topLevelNumber` will identify the level number of the topmost level in the state space EDD.

BoundedSaturationData
+ bound : int + initial : EDDEdge + pruningMethod : PruningMethod + constraint : MDDNode + negConstraint : MDDNode + stateSpace : EDDEdge + topLevelNumber : int {readonly}
+ BoundedSaturation()

Figure B.1: Simplified public interface of the unified bounded saturation algorithm [*a*30]

B.1 Restarting Bounded Saturation

The restarting saturation is a simple iterative version of the EDD-based bounded state space exploration algorithm, using the initial state of the Petri net as initial state for each iteration.

Algorithm B.1: RestartingSaturation

```
   input  : B: int
   output : result of model checking
 1 sd ← new BoundedSaturationData;
 2 sd.pruningMethod ← TruncateExact;                    // sets the pruning method
 3 sd.initial ← EDD representation of Petri net's initial marking;
 4 sd.constraint ← 1;                                              // no constraint
 5 sd.negConstraint ← 0;                                 // no negated constraint
 6 int i ← 1;                                                  // iteration counter
 7 while true do
 8     sd.bound ← i · B;                    // sets the current bound for exploration
 9     sd.BoundedSaturation();
10     ModelChecking(sd.stateSpace.AsMDD());
11     if result is representative to full model then
12         return result of model checking;
13     i ← i + 1;
```

B.2 Continuing Bounded Saturation

As it can be seen, there is only small difference between the restarting saturation (Algorithm B.1) and the continuing saturation (Algorithm B.2). To implement the continuing saturation, the base bounded algorithms had to be extended. The simple saturation-based bounded state space exploration algorithm assumes that the initial state set has exactly one element, which has the encoding $(0, 0, \ldots, 0)$. To support initial state sets with more elements, this special case had to be generalised. The main differences compared to Algorithm B.1 are indicated by "$*$".

Algorithm B.2: ContinuingSaturation

```
   input  : B: int
   output : result of model checking
 1 sd ← new BoundedSaturationData;
 2 sd.pruningMethod ← TruncateExact;                    // sets the pruning method
 3 sd.initial ← EDD representation of Petri net's initial marking;
 4 sd.constraint ← 1;                                              // no constraint
 5 sd.negConstraint ← 0;                                 // no negated constraint
 6 int i ← 1;                                                  // iteration counter
 7 while true do
 8     sd.bound ← i · B;                    // sets the current bound for exploration
 9     sd.BoundedSaturation();
10     ModelChecking(sd.stateSpace.AsMDD());
11     if result is representative to full model then
12         return result of model checking;
*13    // Continue the next iteration from the state space of this iteration.
*14    sd.initial ← sd.stateSpace;
15     i ← i + 1;
```

B.3 Compacting Bounded Saturation

Compacting saturation is a more complex strategy for the B-I-Sat algorithm. The detailed description of the algorithm can be found in Section 2.4. The algorithm relies on the *SubsetAtGivenDistance* function which returns a subset of a given EDD where each encoded tuple has the given value. This is applied to produce the frontier state set of the explored state space. The main differences compared to Algorithm B.1 are indicated by "$*$".

Algorithm B.3: CompactingSaturation

```
   input  : B: int
   output : result of model checking
 1 sd ← new BoundedSaturationData;
 2 sd.pruningMethod ← TruncateExact;                          // sets the pruning method
 3 sd.initial ← EDD representation of Petri net's initial marking;
 4 sd.constraint ← 1;                                                  // no constraint
 5 sd.negConstraint ← 0;                          // no negated constraint at the beginning
 6 int i ← 1;                                                       // iteration counter
* 7 MDDNode M ← 0;
 8 while true do
 9     sd.bound ← i · B;
10     sd.BoundedSaturation();
*11     M ← M ∪ sd.stateSpace.AsMDD();
*12     ModelChecking(M);
13     if result is representative to full model then
14         return result of model checking;
15     else
*16         EDDEdge F ← SubsetAtGivenDistance(sd.stateSpace, incr);      // computes the frontier
*17         sd.initial ← F;    // sets the frontier state set as initial state for next iteration
*18         sd.negConstraint ← M;                        // exclusion of already explored states
```

Algorithm B.4: SubsetAtGivenDistance

input : $\langle v, p\rangle$: EDDEdge, D : int
output : EDDEdge

if $v > D$ **then**
  `// It is impossible to find any states with` $\delta = D$ `in this subgraph.`
  return $\langle \infty, \bot \rangle$;
if $p.level = 0 \wedge v \neq D$ **then**
  `// This path encodes a global state and its distance` $\neq D$.
  return $\langle \infty, \bot \rangle$;
if $p.level = 0 \wedge v = D$ **then**
  `// This path encodes a global state but its distance` $= D$.
  return $\langle v, p \rangle$;
if `CacheFind`$(DISTEQ, \langle v, p\rangle,$ **out** $p')$ **then**
  return $\langle v, p' \rangle$;
EDDNode $n \leftarrow$ `NewNode`$(p.level)$;
foreach $i \in \mathcal{S}_{p.level}$ **do**
  EDDEdge $r \leftarrow$ `SubsetAtGivenDistance`$(\langle v + p[i].value, p[i].node\rangle)$;
  $n[i] \leftarrow \langle r.value - v, r.node\rangle$;
if p *was checked in* **then**
  $n \leftarrow$ `CheckIn`$(p.level, n)$;
`PutIntoCache`$(DISTEQ, \langle v, p\rangle, n)$;
return $\langle v, n \rangle$;

Appendix C

Metamodel of the Intermediate Representations of PLCverif

C.1 Intermediate Model

In the following the metamodel of the intermediate model (IM) language is briefly described, partially based on [*r*24].

The core part of the IM metamodel (the full metamodel without the expression's metamodel) is depicted in Figure C.1. The metamodel of the expressions used in IM can be seen in Figure C.2. In the following, a brief overview can be read about the IM metamodel and its usage.

The root element of the metamodel is **AutomataSystem**. It represents the whole network of automata. This element contains the automata (`automata`), the synchronisations (`synchronisations`) and optionally global variables (`variables`). The *AutomataSystem* has a main automaton (`mainAutomaton`), the automaton that represents the main cycle. The last location of this automaton will be labelled with EoC (meaning "end of PLC cycle"). The execution is started with this automaton. The other automata should wait for synchronisation in their initial location.

The **AutomataInstance** represents a single automaton. It contains locations (`locations`), transitions (`transitions`) and variables (`variables`). The initial location is also set here (`initialLocation`). Semantically there is no difference between the variables of an automaton and the variables of the whole system: even the variables of an automaton can be accessed from every other automaton to match the PLC semantics.

The **Location** represents a location in an automaton. A **Transition** connects two locations (`from` and `to`). It may also have a guard (`guard`) attached, represented by an *AutomataExpression*, and one or more variable assignments (`assignments`). A *Transition* is enabled if its source location is active and its guard condition is evaluated to true. The **Synchronisation** can express a constraint on the firing of transitions. In every step, either *(a)* an active transition fires that has no synchronisation attached, or *(b)* two active transitions fire at the same time if they are connected by a synchronisation.

The transitions can have variable assignments described by a **VariableAssignment** object. It refers to a variable that is modified and an expression that will be the new value of the variable. If there are multiple variable assignments connected to the same transition, their execution order is not defined. The generated models are created in a way that any execution order results in the same final state, no dependent variable assignments are attached to the same transition.

The automata are extended by variables. A **Variable** describes a variable with a given type (`type`) and default value (`defaultValue`). As it was previously stated, a variable can be attached to an au-

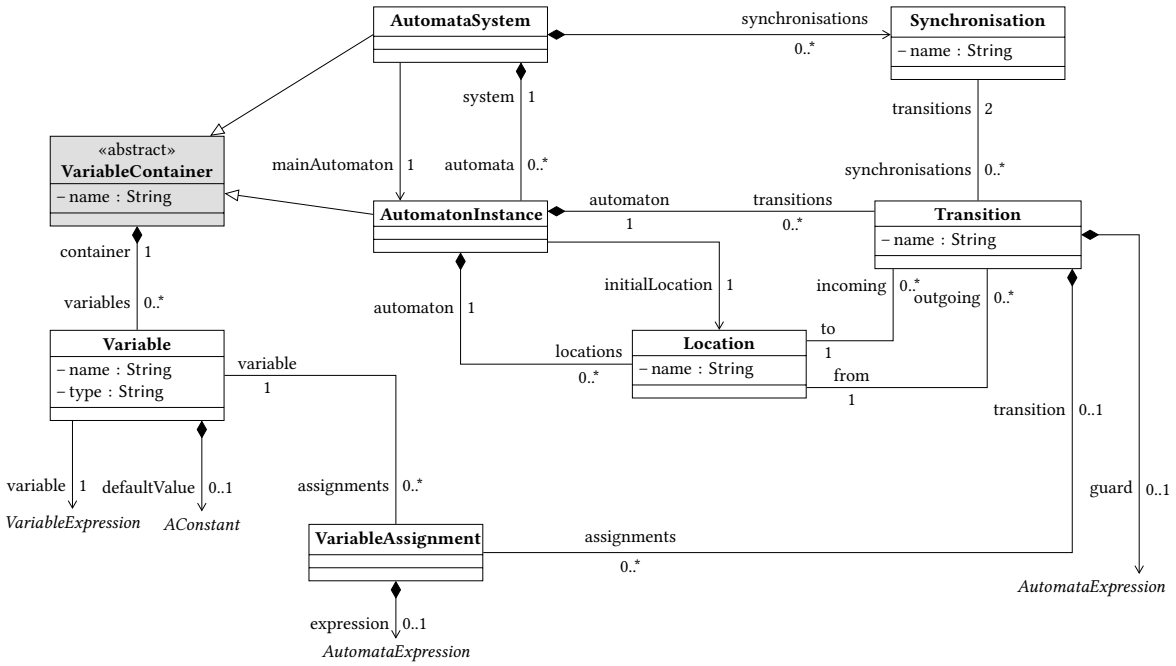


Figure C.1: IM metamodel core

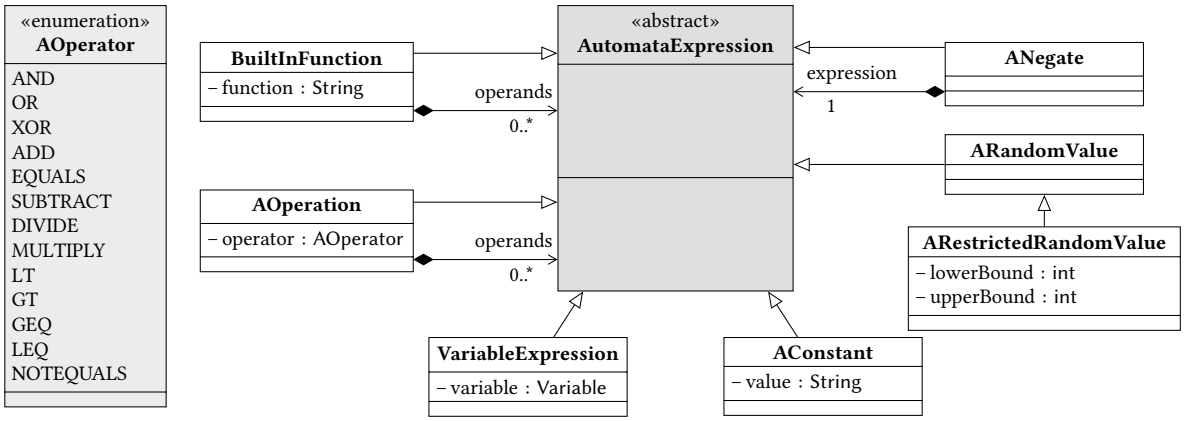


Figure C.2: IM expression metamodel

tomaton or globally to the automata network. The type of the variable is given by a string. The valid type names are those that are defined in the IEC 61131-3 standard [I61131-3].

The expressions used as guard conditions, variable assignments, etc. are all descendants of the abstract **AutomataExpression** class. An expression can be:

- A constant (**AConstant**),
- A variable (**VariableExpression**),
- A built-in PLC function (**BuiltInFunction**),
- A random value of a chosen type (**ARandomValue**) or a random integer value from a given range of numbers (**ARestrictedRandomValue**),
- A negation of any expression (**ANegate**),
- An operation on any Boolean expression (**AOperation**). The possible operators are defined in the **AOperator** enumeration.

C.2 Other Intermediate Representations

To wrap the external model checkers, it is not enough to provide a model checker-independent verification model representation. It is also required to have an independent representation of the output of the model checker which can be used in the report generation. The *GenerationResults* class stores all information that was created during the model generation and verification process, mainly the result, the counterexample and the generation log.

GenerationResults is the main class of this data structure. It contains the abstract syntax tree of the PLC code (`plcAst`), the generated IM model representation (`im`), the result of the verification (`result`), and the counterexample (`counterexample`), if available. This class stores the mapping between the variables in the PLC code, in the IM model and in the concrete syntax representation. For each variable in the automata, a **VariableMapping** is created that stores its different names and references to the corresponding objects. Furthermore, it can store the output of the verification tool written to the standard output (`executionOutput`) or to standard error output (`executionErrorOutput`). It contains the size of the potential state space of the IM model (`pssSize`). Through the `setSetting` and `getSetting` methods, key-value pairs can be stored for each plug-in. The *GenerationResults* also provides a solution for logging. The **GenerationLog** can contain two types of log entries. There will be exactly one **StageLogItem** entry for each stage describing if the stage was successful or not. This is handled by the verification framework. **GenerationLogItem** entries can be added by any plug-in. It can store the stage to which it belongs (`stage`), its severity (`severity`), a message for the user (`message`), and optionally an exception (`exception`). The `time` field will be set automatically.

If the verification tool produces a counterexample (diagnostic trace), the wrapper of the model checker should parse it. A counterexample is represented as a **Counterexample** object. This object can store the name (`name`) and the requirement expression (`expression`) of the counterexample. In addition, it contains one or more **CexStep**s. A *CexStep* describes one step in the counterexample that should represent one PLC cycle, storing the variable values at the end of each PLC cycle. One step consists of many **VariableValuePair**s, each describing a variable (with its internal fully qualified name, `variableFqn`) and its value (`value`).

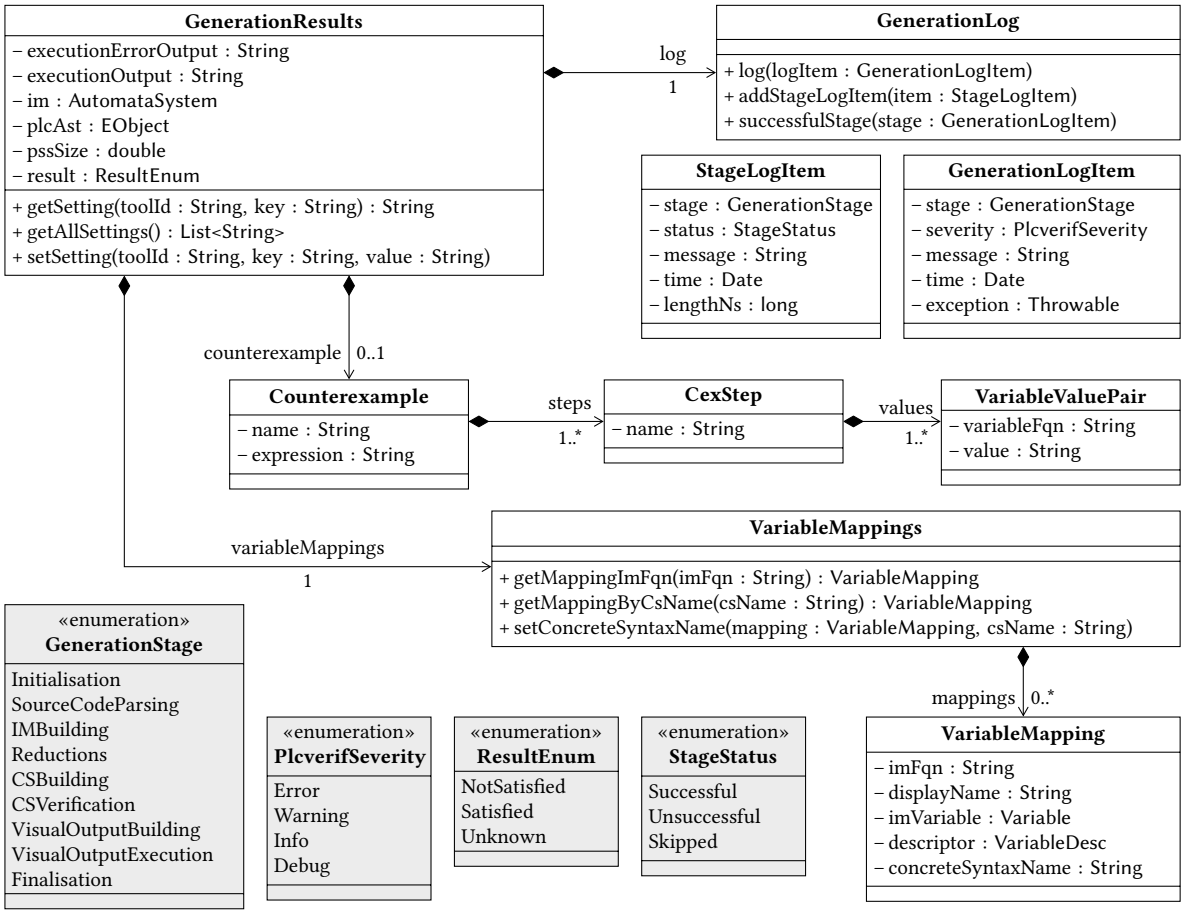


Figure C.3: Metamodel of the generation results and logging structure

Appendix D

Details About the STL to SCLr Translation[1]

D.1 Semantics of the STL Instructions

This part shows an example of determining the semantics of STL instructions. The meaning of the used registers is described informally in Table D.1[2].

We use the A(instruction as an example to present the method. According to the documentation, "*A((AND nesting open) saves the RLO and OR bits and a function code into the nesting stack. A maximum of seven nesting stack entries are possible*" [Sie02]. This instruction does not have any parameters. According to [Sie10] the A(instruction depends on the BR, OR, RLO and nFC status bits and it sets the STA bit to true, and the OR and nFC bits to false. The instruction does not depend on any of the other status bits and it does not modify any other status bits. However, it may depend or affect other data. Based on the informal description, the A(modifies the contents of the nesting stack.

In order to determine the semantics of the A(, all possible combinations of BR, OR, RLO and nFC status bits should be reproduced by a test program, then the resulting values of STA, OR and nFC registers and the nesting stack should be checked. We have generated an STL test code for the A(instruction. The STL code snippet corresponding to a check for a single valuation can be seen in Listing D.1. This specific code snippet can help us to determine the behaviour of the instruction when the BR bit is false, and the OR, RLO and nFC bits are true.

By generating and performing similar checks for all 2^4 combinations, the data in Table D.2 can be obtained. Row 8 of the table was determined by executing the test program in Listing D.1. According to the documentation, the STA, OR and nFC bits are set to constant values unconditionally. However, we have already observed mistakes and contradictions in the official documentations, thus it is worth to check the values of STA, OR and nFC too. In the current case the status bit values observed after the execution matched the defined values, therefore they are omitted from the table. Based on Table D.2 and the documentation the effects of the A(instruction can be summarised:

- It sets the OR and nFC status bits to false (0),
- It sets the STA status bit to true (1),
- It creates a new nesting stack entry, where:
  - The value of OR bit is $\mathsf{OR} \wedge \mathsf{nFC}$,

[1] This chapter is a modified and adapted excerpt from [j1].

[2] Here we omit the registers not necessary for simple STL programs, such as the BR (binary result), OV (overflow), OS (stored overflow) bits and the address registers.

Table D.1: Main registers in Siemens PLCs (based on [Sie98b])

Register	SCLr name	Purpose
RLO	`__RLO`	*Result of last logic operation.*
OR	`__OR`	Helper bit for the "and before or" logical operation (`O` instruction).
nFC	`__NFC`	*Not first computation.* If it is false, the current value of `__RLO` is not taken into account.
STA	`__STA`	*Status bit.* Stores the value of a bit that is referenced.
CC0, CC1	`__CC0`, `__CC1`	*Condition codes.* The result of the last comparison or other operations.
ACCU1, …	`__ACCU*`	*Accumulators.*
nested stack	`__ns*[]`	*Nesting stack.* Temporarily stores register values (`__nsRLO`, `__nsOR`) and the last Boolean operation (`__nsFC*`) while a nested Boolean computation is in progress.

```
1    L   2#00001011   // BR=0, OR=1, RLO=1, nFC=1
2    T   STW
3    A(
4    NOP 0            // place breakpoint here to check the result
5    )                // to restore the empty nesting stack
6    NOP 0
```

Listing D.1: Test code to determine the semantics of the `A(` instruction

Table D.2: Observed behaviour of the `A(` instruction

Before execution				After execution (new nesting stack entry)			
BR	OR	RLO	nFC	nsBR	nsOR	nsRLO	nsFC2,1,0
0	0	0	0	0	0	1	0,0,0
0	0	0	1	0	0	0	0,0,0
0	0	1	0	0	0	1	0,0,0
0	0	1	1	0	0	1	0,0,0
0	1	0	0	0	0	1	0,0,0
0	1	0	1	0	1	0	0,0,0
0	1	1	0	0	0	1	0,0,0
0	1	1	1	0	1	1	0,0,0
1	0	0	0	1	0	1	0,0,0
1	0	0	1	1	0	0	0,0,0
1	0	1	0	1	0	1	0,0,0
1	0	1	1	1	0	1	0,0,0
1	1	0	0	1	0	1	0,0,0
1	1	0	1	1	1	0	0,0,0
1	1	1	0	1	0	1	0,0,0
1	1	1	1	1	1	1	0,0,0

  - The value of RLO bit is $\mathsf{RLO} \vee \neg \mathsf{nFC}$,
  - The value of BR bit is BR[3],
  - The value of function encoding is $(0, 0, 0)$, corresponding to the `A(` instruction [Sie98a], and
- It pushes this new nesting stack entry into the nesting stack.

It is worth noting that contrarily to the straightforward meaning of the description, based on our systematic checks the `A(` instruction does not store the exact values of the RLO and the OR bits (i.e. the nsRLO does not equal to RLO in every case). This demonstrates that the STL to SCLr translation cannot rely only on the informal description of the instructions.

D.2 Identified Correspondences Between STL and SCL

The identified correspondences between STL and SCL [*j*1] are described in Tables D.3, D.4 and D.5.

D.3 Concepts of the Correctness Proof

In this section we present the main concepts of a method to prove the correctness of the translation from STL to SCLr, i.e. if the SCLr equivalents have the same behaviour as the corresponding STL statements according to the experiments. For this, we perform the following steps:

- Defining the formal semantics of SCLr (Section D.3.1),
- Defining the formal semantics of STL (Section D.3.2), and
- Giving a proof strategy to show the equivalence (Section D.3.3).

The following discussion focuses on the principles of this proof strategy and does not provide a complete correctness proof.

D.3.1 Formal Semantics for SCLr

In this section we draw up an operational semantics for the SCL (SCLr) language. We will denote the context of an SCL statement by σ. This is a function $\sigma \colon V \to D$, i.e. a function that assigns a value from pre-defined domains to each defined variable. The program P executed from an initial context σ_0 results in σ_1 which will determine the values of the physical outputs and the initial values of retained variables for the following PLC cycle.

At the beginning of the program execution, each variable has an explicitly or implicitly defined default value (the implicit default values are 0 or false). Then at the beginning of each cycle the non-local variables keep their previously set values, while the initial values of local variables are undefined. The execution ends when the final configuration $(\langle \mathbf{skip;} \rangle, \sigma)$ is reached ("**skip;**" denotes that there is no more program code to be executed).

An intuitive formalisation of the SCL statements' semantics is presented in Figure D.1. This is a small-step semantics, i.e. it defines the operation of a program step by step. The semantics definition in Figure D.1 consists of a set of inference rules. Each inference rule consists of some (zero or more) premises (above the line) and a conclusion (below the line). The operation of a given program with a given initial context can be determined by applying the inference rules one after another until the final configuration $(\langle \mathbf{skip;} \rangle, \sigma)$ is reached.

[3] We were not able to observe directly the BR value of the nesting stack entry. Instead, we have checked the value of the BR status bit after the `)` instruction. The result (i.e. the BR bit of the nesting stack entry equals to the value of the BR status bit) is in accordance with our expectations based on the informal descriptions.

Table D.3: SCLr representation of logical operations

STL	SCLr equivalent
A *var1*	IF __NFC THEN __RLO:=__RLO AND (*var1* OR __OR); ELSE __RLO:=*var1* OR __OR; END_IF; __STA:=*var1*; __NFC:=TRUE;
AN *var1*	IF __NFC THEN __RLO:=__RLO AND (NOT *var1* OR __OR); ELSE __RLO:=NOT *var1* OR __OR; END_IF; __STA:=*var1*; __NFC:=TRUE;
O *var1*	IF __NFC THEN __RLO:=__RLO OR *var1*; ELSE __RLO:=*var1*; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;
ON *var1*	IF __NFC THEN __RLO:=__RLO OR (NOT *var1*); ELSE __RLO:=NOT *var1*; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;
X *var1*	IF __NFC THEN __RLO:=__RLO XOR *var1*; ELSE __RLO:=*var1*; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;
XN *var1*	IF __NFC THEN __RLO:=__RLO XOR (NOT *var1*); ELSE __RLO:=NOT *var1*; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=TRUE;
O	__STA:=TRUE; __OR:=__NFC AND (__OR OR __RLO); __NFC:=__RLO AND __NFC;
= *var1*	IF __MCR THEN *var1*:=__RLO; ELSE *var1*:=FALSE; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;
S *var1*	IF __MCR AND __RLO THEN *var1*:=TRUE; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;
R *var1*	IF __MCR AND __RLO THEN *var1*:=FALSE; END_IF; __OR:=FALSE; __STA:=*var1*; __NFC:=FALSE;

FP *var1*	__OR:=FALSE; __STA:=__RLO; __NFC:=TRUE;
	IF NOT *var1* AND __RLO THEN
	  var1:=__RLO; __RLO:=TRUE;
	ELSE
	  var1:=__RLO; __RLO:=FALSE;
	END_IF;

FN *var1*	__OR:=FALSE; __STA:=__RLO; __NFC:=TRUE;
	IF *var1* AND NOT __RLO THEN
	  var1:=__RLO; __RLO:=TRUE;
	ELSE
	  var1:=__RLO; __RLO:=FALSE;
	END_IF;

NOT	__RLO:=NOT __RLO OR __OR; __STA:=TRUE;
CLR	__RLO:=FALSE; __OR:=FALSE; __STA:=FALSE; __NFC:=FALSE;
SET	__RLO:=TRUE; __OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;
SAVE	__BR:=__RLO;
MCRA	__MCR:=TRUE;
MCRD	__MCR:=FALSE;

Table D.4: SCLr representation of numeric operations

STL	SCLr equivalent
L *var1*	__ACCU2 := __ACCU1; __ACCU1 := *var1*;
T *var1*	IF __MCR THEN *var1*:=__ACCU1; ELSE *var1*:=0; END_IF;

>D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1<__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

>=D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1<=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

<D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1>__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

<=D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1>=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

==D	__OR:=FALSE; __NFC:=TRUE; __RLO:=(__ACCU1=__ACCU2);
	__CC0:=(__ACCU1>__ACCU2); __CC1:=(__ACCU1<__ACCU2);
	__OV:=FALSE; __OR:=FALSE; __NFC:=TRUE; __STA:=__RLO;

Table D.5: SCLr representation of nesting stack operations

STL	SCLr equivalent

A(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=__OR AND __NFC;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

AN(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO OR NOT __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=__OR AND __NFC;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

O(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=FALSE;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=TRUE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

ON(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=FALSE;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=FALSE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=TRUE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

X(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=FALSE;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=TRUE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=FALSE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

XN(	`__nsRLO[7]:=__nsRLO[6]; ... __nsRLO[2]:=__nsRLO[1]; __nsRLO[1]:=__RLO AND __NFC;`
	`__nsOR[7] := __nsOR[6]; ... __nsOR[2] := __nsOR[1]; __nsOR[1] :=FALSE;`
	`__nsBR[7] := __nsBR[6]; ... __nsBR[2] := __nsBR[1]; __nsBR[1] :=__BR;`
	`__nsFC2[7]:=__nsFC2[6]; ... __nsFC2[2]:=__nsFC2[1]; __nsFC2[1]:=TRUE;`
	`__nsFC1[7]:=__nsFC1[6]; ... __nsFC1[2]:=__nsFC1[1]; __nsFC1[1]:=FALSE;`
	`__nsFC0[7]:=__nsFC0[6]; ... __nsFC0[2]:=__nsFC0[1]; __nsFC0[1]:=TRUE;`
	`__OR:=FALSE; __STA:=TRUE; __NFC:=FALSE;`

)	`__OR:=__nsOR[1]; __NFC:=TRUE; __STA:=TRUE; __BR:=__nsBR[1];`
	`IF (NOT __nsFC2[1] AND NOT __nsFC1[1] AND NOT __nsFC0[1])THEN`
	`  __RLO:=(__nsRLO[1] AND __RLO)OR __OR[1]; //A( instruction , FC =(0,0,0)`
	`ELSIF (NOT __nsFC2[1] AND NOT __nsFC1[1] AND __nsFC0[1])THEN`
	`  __RLO:=(__nsRLO[1] AND NOT __RLO)OR __OR[1]; //AN( instruction, FC =(0,0,1)`
	`ELSIF (NOT __nsFC2[1] AND __nsFC1[1] AND NOT __nsFC0[1])THEN`
	`  __RLO:=__nsRLO[1] OR __RLO; //O( instruction , FC =(0,1,0)`
	`ELSIF (NOT __nsFC2[1] AND __nsFC1[1] AND __nsFC0[1])THEN`
	`  __RLO:=__nsRLO[1] OR (NOT __RLO); //ON( instruction, FC =(0,1,1)`
	`ELSIF (__nsFC2[1] AND NOT __nsFC1[1] AND NOT __nsFC0[1])THEN`
	`  __RLO:=__nsRLO[1] XOR __RLO; //X( instruction , FC =(1,0,0)`
	`ELSE`
	`  __RLO:=__nsRLO[1] XOR (NOT __RLO); //XN( instruction, FC =(1,0,1)`
	`END_IF;`
	`__nsRLO[1]:=__nsRLO[2]; ... __nsRLO[6]:=__nsRLO[7]; __nsRLO[7]:=FALSE;`
	`__nsOR[1] := __nsOR[2]; ... __nsOR[6] := __nsOR[7]; __nsOR[7] :=FALSE;`
	`__nsBR[1] := __nsBR[2]; ... __nsBR[6] := __nsBR[7]; __nsBR[7] :=FALSE;`
	`__nsFC2[1]:=__nsFC2[2]; ... __nsFC2[6]:=__nsFC2[7]; __nsFC2[7]:=FALSE;`
	`__nsFC1[1]:=__nsFC1[2]; ... __nsFC1[6]:=__nsFC1[7]; __nsFC1[7]:=FALSE;`
	`__nsFC0[1]:=__nsFC0[2]; ... __nsFC0[6]:=__nsFC0[7]; __nsFC0[7]:=FALSE;`

Note that the expression evaluation is not presented in Figure D.1 in detail (only the `OR` and `AND` operators are defined as illustration), furthermore only the variable assignment and the `IF` statements are presented. In the rules v_1 denotes a variable, e_1, e_2 are expressions, c_1, c_2 are constant values, s_1 and s_2 are SCL statements or SCL statement lists. We distinguish between arithmetic or logic evaluation (denoted by $\longrightarrow_a$) and single-step program evaluations (denoted by $\longrightarrow$). If a program evaluation is possible in more steps we will denote it by $\longrightarrow^*$. Let us denote by $\sigma[v_1 \mapsto c_1]$ the function that is equivalent to σ except that $\sigma(v_1) = c_1$. Formally:

$$\sigma[v_1 \mapsto c_1](x) = \begin{cases} \sigma(x) & \text{if } x \neq v_1 \\ c_1 & \text{if } x = v_1. \end{cases}$$

For the sake of readability, we will use the following comma-separated format too: $\sigma[v_1 \mapsto c_1, \ldots, v_n \mapsto c_n] = \big((\sigma[v_1 \mapsto c_1]) \cdots\big)[v_n \mapsto c_n]$.

$$\frac{\sigma(v_1) = c_1}{(v_1, \sigma) \longrightarrow_a c_1} \text{ SCL Variable value}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1 \qquad (e_2, \sigma) \longrightarrow_a c_2}{(\langle e_1 \texttt{ OR } e_2 \rangle, \sigma) \longrightarrow_a c_1 \vee c_2} \text{ SCL OR expression}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1 \qquad (e_2, \sigma) \longrightarrow_a c_2}{(\langle e_1 \texttt{ AND } e_2 \rangle, \sigma) \longrightarrow_a c_1 \wedge c_2} \text{ SCL AND expression}$$

$$\frac{(\langle s_1 \texttt{;} \rangle, \sigma) \longrightarrow (\langle \textbf{skip}\texttt{;} \rangle, \sigma')}{(\langle s_1 \texttt{;} s_2 \texttt{;} \rangle, \sigma) \longrightarrow (\langle s_2 \texttt{;} \rangle, \sigma')} \text{ SCL Sequence}$$

$$\frac{(e_1, \sigma) \longrightarrow_a c_1}{(\langle v_1 \texttt{:=} e_1 \texttt{;} \rangle, \sigma) \longrightarrow (\langle \textbf{skip}\texttt{;} \rangle, \sigma[v_1 \mapsto c_1])} \text{ SCL Assignment}$$

$$\frac{(e_1, \sigma) \longrightarrow_a \top}{(\langle \texttt{ IF } e_1 \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \texttt{ END\_IF; } \rangle, \sigma) \longrightarrow (s_1, \sigma)} \text{ SCL If (1)}$$

$$\frac{(e_1, \sigma) \longrightarrow_a \bot}{(\langle \texttt{ IF } e_1 \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \texttt{ END\_IF; } \rangle, \sigma) \longrightarrow (s_2, \sigma)} \text{ SCL If (2)}$$

Figure D.1: Simplified SCL semantics

D.3.2 Formal Semantics for STL

In this section the goal is to describe the formal semantics of STL, similarly to the semantics of SCL in the previous section. We will denote the context of an SCL statement by σ, ρ. The first function is $\sigma\colon V \to D$ that assigns a value from pre-defined domains to each defined variable, similarly to the SCL semantics. The second function is $\rho\colon R \to D$ that assigns values to the set of registers $R = \{\mathsf{MCR}, \mathsf{nFC}, \mathsf{RLO}, \mathsf{STA}, \ldots, \mathsf{nsRLO}[1], \mathsf{nsOR}[1], \ldots, \mathsf{nsFC0}[7]\}$.

At the beginning of the program execution, each variable has an explicitly or implicitly defined default value in σ (the implicit default values are 0 or false). Then at the beginning of each cycle the non-local variables keep their previously set values, while the initial values of local variables are undefined. The registers are initialised to their default values at the beginning of each cycle. The default values of the registers are false, except for MCR, RLO and STA which are initialised to true at the beginning of each cycle.

$$\frac{\sigma(v_1) = c_1}{(v_1, \sigma, \rho) \longrightarrow_a c_1}\ \text{STL Variable value}$$

$$\frac{\rho(r_1) = c_1}{(r_1, \sigma, \rho) \longrightarrow_a c_1}\ \text{STL Register value}$$

$$\frac{(\langle s_1 \rangle, \sigma, \rho) \longrightarrow (\langle \textbf{skip} \rangle, \sigma', \rho')}{(\langle s_1 \quad s_2 \rangle, \sigma, \rho) \longrightarrow (\langle s_2 \rangle, \sigma', \rho')}\ \text{STL Sequence}$$

Figure D.2: Simplified base STL semantics

We define the basics of semantics, such as the variable and register values, or the semantics of the sequence of instructions, as follows in Figure D.2. In the rules v_1 is a variable, c_1 is a constant value, s_1 and s_2 are STL statements or STL statement lists.

Formalising the discovered STL semantics. As it was discussed previously, the semantics of the STL instructions are not defined precisely. In our method we conduct systematic experiments to determine the semantics of the STL instructions in each possible configuration. The STL semantics is known only through these observed semantics. This was summarised in tables, similarly to Table D.2. These tables can be systematically transformed into inference rules. Each row of the table can be represented as a single inference rule, therefore the semantics of a given STL instruction will be formalised as a set of inference rules, one for each possible initial configuration. This is demonstrated by the following example.

Example. *Here we will use a simple "or" operation* `O v1` *as an example, where* `v1` *is a variable. The observed semantics of* `O v1` *, obtained through a series of tests as described before, can be seen in Table D.6.*

Table D.6: Observed STL semantics for `O v1`

Before execution			**After execution**				
RLO	nFC	v1	v1	OR	STA	RLO	nFC
0	0	0	0	0	0	0	1
0	0	1	1	0	1	1	1
0	1	0	0	0	0	0	1
0	1	1	1	0	1	1	1
1	0	0	0	0	0	0	1
1	0	1	1	0	1	1	1
1	1	0	0	0	0	1	1
1	1	1	0	0	1	1	1

Each row of this table describes the semantics of the `O v1` *statement with different preconditions. For example, row 2 defines the following formal semantics:*

$$\frac{\rho(\mathsf{RLO}) = \bot \quad \rho(\mathsf{nFC}) = \bot \quad \sigma(v_1) = \top}{(\langle \mathtt{O}\ v_1 \rangle, \rho, \sigma) \longrightarrow (\langle \textbf{skip} \rangle, \rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top], \sigma)}\ \text{STL O } v_1 \text{ (2)}$$

After formalising the SCL and STL semantics, the only remaining step is to prove that the suggested SCLr representations of the STL statements will provide the same results. This proof will be drawn up in the next section.

```
IF __NFC THEN
    __RLO:=__RLO OR v1;
ELSE
    __RLO:=v1;
END_IF;
__OR:=FALSE;
__STA:=v1;
__nFC:=TRUE;
```

Listing D.2: SCLr equivalent of the `O v1` STL statement

D.3.3 Strategy for the Correctness Proof

Now it is possible to define formally the correctness of the STL to SCL translation. Formally, the goal is to prove the following:

$$\big((P_{STL}, \sigma_1, \rho_1) \longrightarrow^* (\langle \textbf{skip} \rangle, \sigma_2, \rho_2)\big) \Longrightarrow \big((P_{SCL}, \sigma_1') \longrightarrow^* (\langle \textbf{skip}\,; \rangle, \sigma_2')\big),$$

where P_{SCL} is the SCLr representation of the STL code P_{STL}, and σ_i' is the representation of σ_i, ρ_i such that:

$$\sigma_i'(x) = \begin{cases} \sigma_i(x) & \text{if } x \text{ is a real variable} \\ \rho_i(y) & \text{if } x \text{ is the SCLr variable representing the register } y \text{ (\_y=}x\text{).} \end{cases}$$

The program P_{STL} is a sequence of STL statements. Based on the SCL Sequence and STL Sequence inference rules of the semantics definitions discussed before, it is enough to prove that the behaviour of each STL instruction corresponds to their SCLr representations' behaviour. We have to show that the proposed SCLr equivalent of a certain STL instruction provides the same semantics as the original STL instruction. As the semantics of a given STL instruction is formalised as a set of inference rules (as discussed in the previous section), the goal is to show that for each STL inference rule given the same premises (equivalent initial contexts), given the SCLr representation, and using the inference rules of the SCL semantics, the reached final configuration of the SCLr program corresponds to the final configuration of the STL instruction. This is demonstrated by the following example.

Example. *Based on Table D.6, the SCLr equivalent presented in Listing D.2 can be proposed for* `O v1`.

For each row of Table D.6 it is possible to formally prove based on the defined SCLr semantics that the proposed SCLr equivalent provides the same result. The inference tree in Figure D.3 proves that the above SCLr code matches the previously discussed STL O v_1 (2) semantic rule. This proof shows that from the same premises ($\sigma(\_\mathsf{nFC}) = \bot, \sigma(\_\mathsf{RLO}) = \bot, \sigma(v_1) = \top$), by applying the inference rules of the SCL semantics, the reached configuration corresponds to the one reached by the execution of the STL statement: the context $\sigma'[\_\mathsf{OR} \mapsto \bot, \_\mathsf{STA} \mapsto \top, \_\mathsf{RLO} \mapsto \top, \_\mathsf{nFC} \mapsto \top]$ reached by the SCLr code emulates the final configuration in the STL semantics definition $(\rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top], \sigma)$.

It can also be seen in Figure D.3 that the result does not depend on the value of the RLO register, therefore the same proof can be used for the STL O v_1 (6) semantic rule, describing row 6 of Table D.6:

$$\frac{\rho(\mathsf{RLO}) = \top \quad \rho(\mathsf{nFC}) = \bot \quad \sigma(v_1) = \top}{(\langle \texttt{O}\ v_1 \rangle, \rho, \sigma) \longrightarrow (\langle \textbf{skip} \rangle, \rho[\mathsf{OR} \mapsto \bot, \mathsf{STA} \mapsto \top, \mathsf{RLO} \mapsto \top, \mathsf{nFC} \mapsto \top], \sigma)}\ \text{STL O } v_1 \text{ (6)}.$$

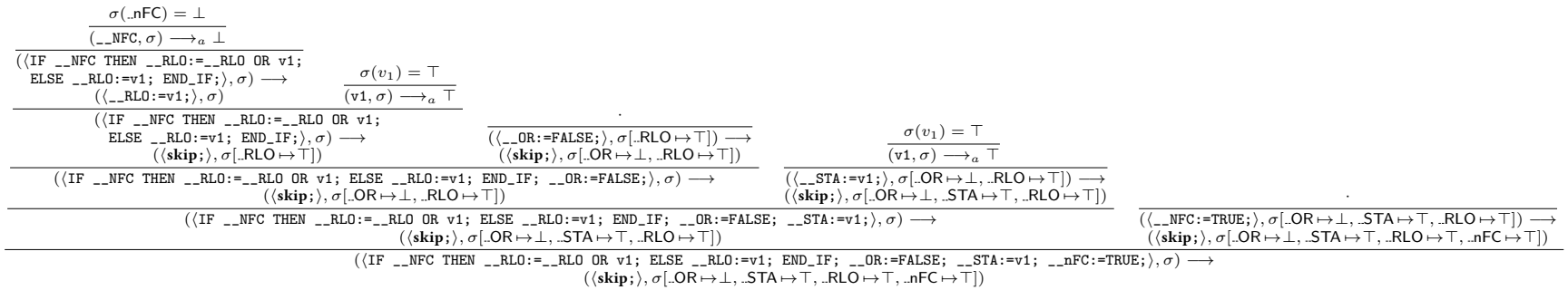


Figure D.3: Proof of semantic equivalence between the SCLr representation of `O v1` and the semantic rule STL O v_1 (2)

Appendix E

Semantics of PLCspecif

This chapter provides a brief overview of PLCspecif's formal semantics. The description in this chapter is an excerpt of the own work [r22], which provides a more complete syntax and semantics definition.

The formal semantics of PLCspecif is defined as a construction of a variant of the widely-known timed automata formalism. Timed automaton has a well-defined formal semantics, and it is a relatively high-level formalism, thus it is convenient for describing the formal semantics of PLCspecif. Furthermore, the semantic of a specification defined as an automaton is close to the control flow graph of its implementation, helping to design a code generator that follows the formal semantics.

E.1 Timed Automata

A timed automaton [BY04] is essentially a set of locations, clocks, and edges (transitions), each having a source and target location, and a clock constraint. To make the usage of the formalism more convenient, we extended the TA definition with data variables that can be used in conditions of edges, also firing transitions represented by the edges can assign new values to data variables. The semantics definition is based on the timed automaton defined below, which is an extension of the timed automata formalism defined in [BY04]. These extensions do not increase the expressive power of the formalism, but simplify the discussion.

Definition E.1 (Timed automaton with data variables (extended, based on [BY04])). A timed automaton (TA) is a tuple $TA = \langle L, \ell_0, C, A, E, I, V, v_0, U \rangle$ where:

- L is a set of locations;
- $\ell_0 \in L$ is the initial location;
- C is the set of clocks;
- A is a set of actions, co-actions and the internal τ-action;
- $E \subseteq L \times A \times B_{bool}(C) \times B_{bool}(V) \times (V \to (B'(V) \cup \{\cdot\})) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset (if $(\ell, a, cg, vg, m, r, \ell') \in E$ then it is denoted as $\ell \xrightarrow{a,cg,vg,m,r} \ell'$, where $a \in A$ is the action, $cg \in B_{bool}(C)$ is the guard on clocks, $vg \in B_{bool}(V)$ is the guard on data variables, $m \colon V \to (B'(V) \cup \{\cdot\})$ is the change of data variable values (where $\cdot$ means "no change"), and $r \in 2^C$ is the set of clocks to reset);
- $I \colon L \to B_{bool}(C)$ assigns invariants to locations;

- V is the set of data variables;
- $v_0 : V \to \mathcal{D}$ assigns initial values to the variables ($\mathcal{D}$ is the set of possible variable values);
- $U \subseteq L$ is the set of urgent locations.

N.b.: $B_{bool}(C)$ is a clock constraint of form $x \sim n$ or $x - y \sim n$ for $x, y \in C, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. $B'(V)$ is an expression on data variables in V and/or constants. $B_{bool}(V) \subset B'(V)$ is a Boolean expression using data variables in V and/or constants. ▪

The operation semantics defined for this timed automaton is the following:

Definition E.2 (Semantics of TA with data variables (based on [BY04])).
Let $\langle L, \ell_0, C, A, E, I, V, v_0, U \rangle$ be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \to \rangle$, where:

- $S \subseteq L \times \mathbb{R}^C \times (V \to \mathcal{D})$ is the set of states;
- $s_0 = (\ell_0, u_0, v_0) \in S$ is the initial state;
- $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation such that:
  - (delay transition) $(\ell, u, v) \xrightarrow{d} (\ell, u + d, v)$ if $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(\ell)$ and $\ell \notin U$,
  - (action transition) $(\ell, u, v) \xrightarrow{a} (\ell', u', v')$ if there exists $\ell \xrightarrow{a,cg,vg,m,r} \ell'$ in E such that $u \in cg$, $v \in cg$, $v' = m(v)$, $u' = [r \rightarrowtail 0]u$, and $u' \in I(\ell')$,

  where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock $x \in C$ to the value $u(x) + d$, and $u' = [r \rightarrowtail 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$. The notation $v \in cg$ means cg guards are evaluated to true having the v data variable values. v' will be the evaluation of m while having current data variable values as defined in v. For each variable $i \in V$, the definition of $v' = m(v)$ is the following:

$$(m(v))(i) = v'(i) = \begin{cases} v(i) & \text{if } m(i) = \cdot \\ m(i) & \text{else} \end{cases}$$

▪

If the timed automaton has only the internal action τ, then the semantics can be defined based on a Kripke structure instead of a labelled transition system (LTS). The set of states, initial states and transitions are the obvious translations of the ones defined for LTS. The labelling for each state $s = (\ell, u, v)$ should (at least) contain all the atomic propositions that are true for the variables u, v. Obviously, there can be a large number of these atomic propositions. It has to be noted that in most cases the time-related features of the defined formalism are not used, only if a precise and consistent modelling is required for verification purposes.

As the formal semantics definition is presented as a pseudocode, we provide a (reduced) metamodel of the timed automaton corresponding to the definition above. The metamodel can be found in Figure E.1.

E.2 Translation Algorithms

This section presents the precise pseudocode of the transformation from PLCspecif modules to (timed) automata. The algorithms are illustrated by non-formal figures to help the reader to understand the main ideas. In the following we focus only on the translation of state machine modules. For this, two main algorithms are shown:

- The general representation for any module:
  TranslateRec(Module, TALocation, TALocation, **inout** TA) – Algorithm E.1, and

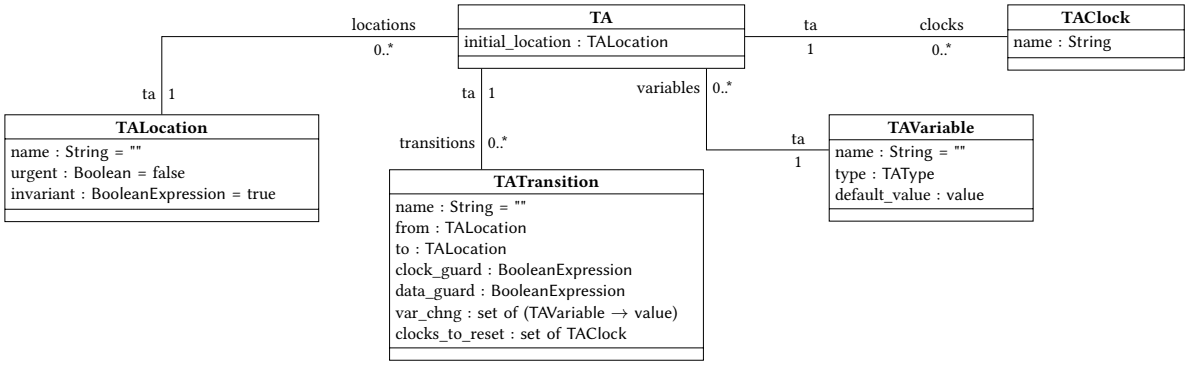


Figure E.1: Metamodel of the timed automata

- The specific representation of state machine modules:
  TranslateRecSpec(StatemachineModule, TALocation, TALocation, **inout** TA) – Algorithm E.2

These functions are illustrated in Figure E.2.

Furthermore, some utility functions used in the presented pseudocode are also given. The representation of other module types can be found in [*r*22]. The report [*r*22] also contains the full metamodel of PLCspecif, of which an excerpt was shown in Figures 4.1 and 4.3. Please note that the following pseudocode uses a simplified notation for building expression trees. For example, $a \vee b$ denotes the current value of the expression, while $\langle a \vee b \rangle$ denotes the expression tree encoding $a \vee b$, and the variables a and b refer to variables in the timed automata (thus their values are not known at translation time).

Algorithm E.1: TranslateRec

input : m : Module, ℓ_1, ℓ_5 : TALocation
inout : a : TA // The TA representation

1 // This algorithm translates the modules recursively. The representation will be inserted between the two given TA locations (ℓ_1, ℓ_5).
2 // Creating intermediate locations
3 $\ell_4 \leftarrow$ *new* TALocation*(ta: a)*;
4 // Translating input definitions
5 $\ell_{from} \leftarrow \ell_1$;
6 **foreach** $vde \in m$.O_inputDefinitions **do**
7 $\quad \ell_{to} \leftarrow$ *new* TALocation*(ta: a)*;
8 $\quad$ *new* TATransition*(from:* ℓ_{from}*, to:* ℓ_{to}*, clock_guard:* $\emptyset$*, data_guard:* $\emptyset$*, var_chng:* $\langle$mapping(vde.variable, TAVariable) := translate(vde.expression)$\rangle$*, ta: a)*;
9 $\quad \ell_{from} \leftarrow \ell_{to}$;
10 $\ell_2 \leftarrow \ell_{from}$;
11 // Translating event definitions and their computations (only for LeafModules)
12 **if** m ***is a*** LeafModule **then**
13 $\quad \ell_{from} \leftarrow \ell_2$;
14 $\quad$ **foreach** $e \in m$.events **do**
15 $\quad\quad \ell_{to} \leftarrow$ *new* TALocation*(ta: a)*;
16 $\quad\quad v \leftarrow$ *new* TAVariable*(name: e*.name*, type:* bool*, ta: a)*;
17 $\quad\quad P \leftarrow \{e' \in m$.events : e'.priority $< e$.priority$\}$; // Higher priority events
18 $\quad\quad h \leftarrow \langle \bigwedge_{e' \in P} \neg e'$.triggerExpression$\rangle$; // Higher priority events are not active
19 $\quad\quad$ *new* TATransition*(from:* ℓ_{from}*, to:* ℓ_{to}*, clock_guard:* $\emptyset$*, data_guard:* $\emptyset$*, var_chng:* $\langle v := (e$.triggerExpression $\wedge\ h)\rangle$*, ta: a)*;
20 $\quad\quad$ mapping(e, TAVariable) $\leftarrow v$;
21 $\quad\quad \ell_{from} \leftarrow \ell_{to}$;
22 $\quad \ell_3 \leftarrow \ell_{from}$;
23 **else**
24 $\quad \ell_3 \leftarrow$ *new* TALocation*(ta: a)*;
25 $\quad$ *new* TATransition*(from:* ℓ_2*, to:* ℓ_3*, clock_guard:* $\emptyset$*, data_guard:* $\emptyset$*, var_chng:* $\emptyset$*, ta: a)*;
26 // Module-specific translation (different for each type)
27 TranslateRecSpec(m, ℓ_3, ℓ_4, a);
28 // Translating output definitions
29 $\ell_{from} \leftarrow \ell_4$;
30 **foreach** $vde \in m$.O_outputDefinitions **do**
31 $\quad \ell_{to} \leftarrow$ *new* TALocation*(ta: a)*;
32 $\quad$ *new* TATransition*(from:* ℓ_{from}*, to:* ℓ_{to}*, clock_guard:* $\emptyset$*, data_guard:* $\emptyset$*, var_chng:* $\langle vde$.variable := vde.expression$\rangle$*, ta: a)*;
33 $\quad \ell_{from} \leftarrow \ell_{to}$;
34 *new* TATransition*(from:* ℓ_{from}*, to:* ℓ_5*, clock_guard:* $\emptyset$*, data_guard:* $\emptyset$*, var_chng:* $\emptyset$*, ta: a)*;

Algorithm E.2: TranslateRecSpec

input : m : StatemachineModule, ℓ_1, ℓ_4 : TALocation
inout : a : TA // The TA representation

// This algorithm translates a StatemachineModule module recursively. The representation will be inserted between the two given TA locations.

// Creating intermediate locations
$\ell_2 \leftarrow$ *new* TALocation*(ta: a)*;
$\ell_3 \leftarrow$ *new* TALocation*(ta: a)*;

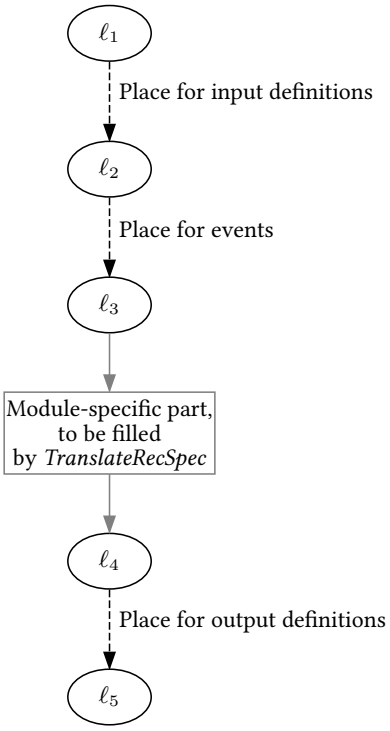
// Creating TA variables
$activeState \leftarrow$ *new* TAVariable*(type:* enum of `basicStatesIn`(m)*, defaultValue:* m.initialState*, ta: a)*;
foreach $s \in$ `allPseudoStatesOf`$(m$.rootState$)$ **do**
  if s **is a** DeepHistoryState **then**
    $v \leftarrow$ *new* TAVariable*(type:* enum of `basicStatesIn`(m)*, defaultValue:* s.defaultState*, ta: a)*;
    `mapping`(s,TAVariable) $\leftarrow v$;

$B \leftarrow$ *false*;
// Translating non-triggered transitions as TA transitions
foreach $t \in m$.transitions **do**
  if t.trigger $= \emptyset$ **then**
    $srcActive \leftarrow$ `sourceActive`$(t, activeState)$;
    // Collecting the potential conflicting transitions (where the priority would disable t)
    $conf \leftarrow \{t' \in m.\text{transitions} \mid t' \neq t \wedge$ `basicStatesOf`$(t'.\text{from}) \cap$ `basicStatesOf`$(t.\text{from}) \neq \emptyset \wedge t'.\text{priority} < t.\text{priority}\}$;
    $higherPriorityDisables \leftarrow \langle \bigvee_{t' \in conf}$ `sourceActive`$(t', activeState) \wedge t'.\text{guard}\rangle$;
    $g \leftarrow \langle srcActive \wedge t.\text{guard} \wedge \neg higherPriorityDisables\rangle$; // t can fire if the source is active and guard is true and no conflicting transition with higher priority disables it
    $B \leftarrow \langle B \vee g\rangle$; // B is a symbolic expression (not evaluated in transformation time)
    // After firing the target state will be activated
    if $\neg$(t.to **is a** DeepHistoryState) **then**
      $vc \leftarrow \langle activeState :=$ `literal`$(t.\text{to})\rangle$;
      foreach $h \in$ `historyStatesToUpdate`$(t.\text{to})$ **do**
        // $\circ$ is the concatenation operator, e.g. $(a_1, \ldots, a_n) \circ (b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$.
        $vc \leftarrow vc \circ \langle$`mapping`$(h$, TAVariable$) :=$ `literal`$(t.\text{to})\rangle$; // Saving history
    else
      $vc \leftarrow \langle activeState :=$ `mapping`$(t.\text{to}$, TAVariable$)\rangle$; // Restoring state from history
    $t_1 \leftarrow$ *new* TATransition*(from: ℓ_1, to: ℓ_1, clock_guard: $\emptyset$, data_guard: g, var_chng: vc, ta: a)*;
    $t_2 \leftarrow$ *new* TATransition*(from: ℓ_3, to: ℓ_3, clock_guard: $\emptyset$, data_guard: g, var_chng: vc, ta: a)*;

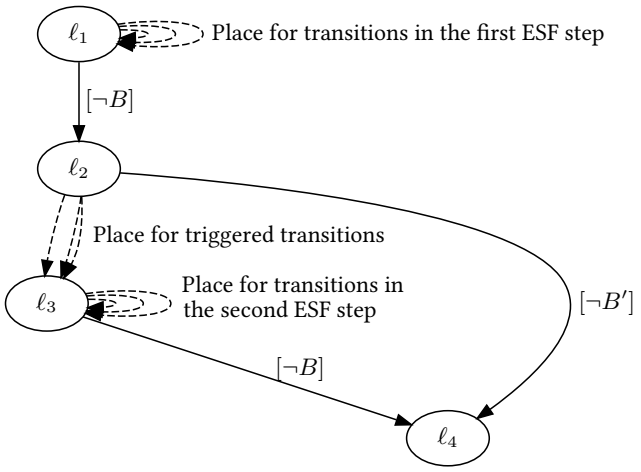
// Translating triggered transitions as TA transitions
$B' \leftarrow$ *false*;
foreach $t \in m$.transitions **do**
  if t.trigger $\neq \emptyset$ **then**
    $srcActive \leftarrow$ `sourceActive`$(t, activeState)$;
    $g \leftarrow \langle srcActive \wedge$ `mapping`$(t.\text{trigger}$, TAVariable$) \wedge t.\text{guard}\rangle$; // t can fire if the source is active and guard is true and the event is triggered
    $B' \leftarrow \langle B' \vee g\rangle$; // B' is a symbolic expression (not evaluated in transformation time)
    // After firing the target state will be activated
    if $\neg$(t.to **is a** DeepHistoryState) **then**
      $vc \leftarrow \langle activeState :=$ `literal`$(t.\text{to})\rangle$;
      foreach $h \in$ `historyStatesToUpdate`$(t.\text{to})$ **do**
        // $\circ$ is the concatenation operator, e.g. $(a_1, \ldots, a_n) \circ (b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$.
        $vc \leftarrow vc \circ \langle$`mapping`$(h$, TAVariable$) :=$ `literal`$(t.\text{to})\rangle$; // Saving history
    else
      $vc \leftarrow \langle activeState :=$ `mapping`$(t.\text{to}$, TAVariable$)\rangle$; // Restoring state from history
    $t_3 \leftarrow$ *new* TATransition*(from: ℓ_2, to: ℓ_3, clock_guard: $\emptyset$, data_guard: g, var_chng: vc, ta: a)*;

// If none of the triggered transition could fire, skip this phase and the second ESF step
new TATransition*(from: ℓ_2, to: ℓ_4, clock_guard: $\emptyset$, data_guard: $\neg B'$, var_chng: vc, ta: a)*;

// When the exhaustive stabilisation firing step is over
new TATransition*(from: ℓ_1, to: ℓ_2, clock_guard: $\emptyset$, data_guard: $\neg B$, var_chng: $\langle\rangle$, ta: a)*;
new TATransition*(from: ℓ_3, to: ℓ_4, clock_guard: $\emptyset$, data_guard: $\neg B$, var_chng: $\langle\rangle$, ta: a)*;



(a) Illustration of the function *TranslateRec* (Algorithm E.1)



(b) Illustration of the function *TranslateRecSpec* for StatemachineModules (Algorithm E.2)

Figure E.2: Illustration of the translation functions

Algorithm E.3: sourceActive

```
1  // This algorithm returns a Boolean expression that is true iff the source state (or any of its
       basic states, if it is compound) of the given transition is active.
   input  : t : Transition, activeState : TAVariable
   output : b : TAExpression
2  return ⟨⋁_{s' ∈ basicStatesOf(t.from)} (activeState = literal(s'))⟩;
```

Algorithm E.4: allLeafmodulesOf

```
1  // Collects recursively all leaf modules of m, including m if it is a LeafModule.
   input  : m : Module
   output : set of Module
2  if m is a LeafModule then
3  |   return {m};
4  else if m is a AlternativeModule then
5  |   return {allLeafmodulesOf(m.onTrue), allLeafmodulesOf(m.onFalse)};
6  else if m is a CompositeModule then
7  |   SM ← ∅;
8  |   foreach m' ∈ m.submodules do
9  |   |   SM ← SM ∪ allLeafmodulesOf(m');
10 |   return SM;
11 return error;
```

Algorithm E.5: allParentStatesOf

```
// Collects all container CompositeStates of s, not including s itself.
input  : s : AbstractState
output : PS : set of CompositeState
PS ← ∅;
s' ← s;
while s'.containerState ≠ undefined do
    PS ← PS ∪ s'.containerState;
    s' ← s'.containerState;
```

Algorithm E.6: allPseudoStatesOf

```
input  : s : AbstractState
output : set of PseudoState
PS ← ∅;
if s is a PseudoState then
    PS ← {s};
else if s is a CompositeState then
    foreach s' ∈ s.containedStates do
        PS ← PS ∪ allPseudoStatesOf(s');
return PS;
```

Algorithm E.7: basicStatesIn

```
// Collects all basic states defined for a given StatemachineModule m.
input  : m : StatemachineModule
output : set of BasicState
return basicStatesOf(m.rootState);
```

Algorithm E.8: basicStatesOf

```
input  : s : AbstractState
output : set of BasicState
AS ← ∅;
if s is a BasicState then
    AS ← {s};
else if s is a CompositeState then
    foreach s' ∈ s.containedStates do
        AS ← AS ∪ basicStatesOf(s');
return AS;
// If s is a PseudoState, AS remains empty.
```

Algorithm E.9: historyStatesToUpdate

```
input  : s : AbstractState                          // s is the newly activated state
output : PS : set of PseudoState
PS ← ∅;
foreach m ∈ allParentStatesOf(s) do
    foreach s' ∈ m.containedStates do
        if s' is a DeepHierarchyState then
            PS ← PS ∪ s;
```

E.3 Mapping from PLCspecif Semantics to IM

Table E.1 briefly describes the correspondence between the automata formalism used for the semantics definition of PLCspecif and the intermediate model used in PLCverif. The table uses the naming of the metamodels presented in Figure E.1 (timed automaton metamodel) and Figure C.1 (IM metamodel). As it was discussed earlier, the timed parts of the TA used to describe the PLC timers are not used for the IM representation. Instead, the PLC timer modules are represented using the principles introduced in Section 3.3.1.

Table E.1: Correspondence between the TA and IM metamodels

TA element	**Corresponding IM element(s)**
TA – `initial_location`	AutomatonInstance (`mainAutomaton` in an AutomataSystem) – `initialLocation` (Location corresponding to the TA's `initial_location`)
TAVariable – `name` – `type` – `default_value`	Variable (contained by the single automaton instance) – `name` – `type` (corresponding type) – `defaultValue` (AConstant describing the TA's `default_value`)
TALocation – `name` – `urgent` – `invariant`	Location (contained by the single automaton instance) – `name` *not used in non-timed models* (always false) *not used in non-timed models* (always true)
TATransition – `name` – `from` – `to` – `clock_guard` – `data_guard` – `var_chng` – `clocks_to_reset`	Transition – `name` – `from` (Location corresponding to the TA's `from` TALocation) – `to` (Location corresponding to the TA's `to` TALocation) *not used in non-timed models* – `guard` (AutomataExpression describing the TA's `data_guard`) – `assignments` (one or more equivalent VariableAssignments) *not used in non-timed models*
TAClock – `name`	*not used in non-timed models* *not used in non-timed models*

Appendix F

List of Abbreviations

ALICE	A Large Ion Collider Experiment (LHC experiment at CERN)
AST	abstract syntax tree
ATLAS	A Toroidal LHC Apparatus (LHC experiment at CERN)
BDD	binary decision diagram
BFS	breadth-first search
BPCS	basic process control systems
BUTE	Budapest University of Technology and Economics
CERN	European Organization for Nuclear Research
CFA	control flow automaton
CFG	control flow graph
CMS	Compact Muon Solenoid (LHC experiment at CERN)
COI	cone of influence reduction
CPC	continuous process control
CPN	coloured Petri net
CPU	central processing unit
CSP	communicating sequential processes
CTL	computation tree logic
DFS	depth-first search
EDD	edge-valued decision diagram
ESF	exhaustive stabilisation firing
FBD	Function Block Diagram (IEC 61131 PLC programming language)
FMS	flexible manufacturing system (benchmark Petri net model)
HDD	hard disk drive
ICS	industrial control systems
IEC	International Electrotechnical Commission
IL	Instruction List (IEC 61131 PLC programming language)
IM	intermediate model
ISO	International Organization for Standardization
ISOLDE	Isotope Mass Separator On-Line Facility (radioactive ion beam facility at CERN)
LAD	Ladder Diagram (Siemens PLC programming language)
LD	Ladder Diagram (IEC 61131 PLC programming language)
LHC	Large Hadron Collider (particle collider at CERN)

LTL	linear temporal logic
LTS	labelled transition system
MDD	multivalued decision diagram
PC	personal computer
PLC	programmable logic controller
PLCSE	PLC pour la sécurité, safety PLC
PN	Petri net
PRISE	primary-to-secondary leakage (malfunction of a nuclear power plant)
PSL	property specification language
PSS	potential state space
PVS	Prototype Verification System (specification language)
RLO	result of last logic operation (status bit of Siemens PLCs)
RSML	Requirements State Machine Language (specification language)
RSS	reachable state space
SAT	Boolean satisfiability problem
SCADA	supervisory control and data acquisition
SCL	Structured Control Language (Siemens PLC programming language)
SCLr	Structured Control Language with explicit register representations
SFC	Sequential Function Chart (IEC 61131 / Siemens PLC programming language)
SIL	safety integrity level
SR	slotted ring (benchmark Petri net model)
ST	Structured Text (IEC 61131 PLC programming language)
STL	Statement List (Siemens PLC programming language)
TA	timed automaton
TCAS	Traffic Collision Avoidance System (safety system in aviation)
TL	temporal logic
TOF	off-delay timer (IEC 61131 standard timer function block)
TON	on-delay timer (IEC 61131 standard timer function block)
TP	pulse timer (IEC 61131 standard timer function block)
UML	Unified Modeling Language
UNICOS	Unified Industrial Control System
VDM-SL	Vienna Development Method Specification Language (specification language)

Publications

Number of publications:	29
Number of peer-reviewed journal papers (written in English):	6
Number of articles in journals indexed by WoS or Scopus:	6
Number of publications (in English) with at least 50% contribution of the author:	3
Number of peer-reviewed publications:	22
Number of independent citations:	23

Publications Linked to the Theses

	Journal papers	International conference and workshop papers	Local events	Technical reports
Thesis 1	[*j*2],[*j*4]	[*c*10],[*c*17],[*c*18]	[*e*20],[*e*21]	—
Thesis 2	[*j*1],[*j*3]	[*c*8]*,[*c*9]*,[*c*11],[*c*13],[*c*14],[*c*15],[*c*16]	—	[*r*23],[*r*24]
Thesis 3	—	[*c*5],[*c*6],[*c*7],[*c*8]*,[*c*9]*,[*c*12]	[*e*19]	[*r*22]

* These publications are attached to multiple theses.
This classification follows the faculty's Ph.D. publication score system.

Journal Papers

[*j*1] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. PLC program translation for verification purposes. *Periodica Polytechnica, Electrical Engineering and Computer Science*, 2017. URL: http://mit.bme.hu/~darvas/publications/PerPol2017_DarvasEtAl.pdf. Accepted, under publication.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*j*2] Dániel Darvas, András Vörös, and Tamás Bartha. Improving saturation-based bounded model checking. *Acta Cybernetica* 22(3), 2016, pp. 573–589. DOI: 10.14232/actacyb.22.3.2016.2.

▷ *Own contribution, joint paper with M.Sc. thesis supervisors.*

[*j*3] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor M. González Suárez. Applying model checking to industrial-sized PLC programs. *IEEE Transactions on Industrial Informatics* 11(6), 2015, pp. 1400–1410. DOI: 10.1109/TII.2015.2489184. $IF_{2014} = 8.78$.

▷ *The intermediate representation of the SFC language and the intermediate model is my contribution. The PLC behaviour description and the variable abstraction method are contributions of B. Fernández Adiego. The rest of the verification method is a joint work with B. Fernández Adiego. The contribution of J.O. Blech is the discussion about the correctness of the approach. E. Blanco Viñuela, J-C. Tournier, S. Bliudze and V.M. González Suárez were helping the work as advisors.*

[*j*4] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation-based CTL model checking. *Proceedings of the Estonian Academy of Sciences* 62(1), 2013, pp. 59–70. DOI: 10.3176/proc.2013.1.07. $IF_{2013} = 0.37$.

▷ *Own contribution, joint paper with M.Sc. thesis supervisors.*

International Conference and Workshop Papers

[*c*5] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Well-formedness and invariant checking of PLCspecif specifications. In: *Proceedings of the 24th PhD Mini-Symposium*, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2017. In press.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*c*6] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Conformance checking for programmable logic controller programs and specifications. In: *11th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 29–36. IEEE, 2016. DOI: 10.1109/SIES.2016.7509409.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*c*7] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. PLC code generation based on a formal specification language. In: *14th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 389–396. IEEE, 2016. URL: http://mit.bme.hu/~darvas/publications/INDIN2016_DarvasEtAl.pdf.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*c*8] Dániel Darvas. Practice-oriented formal methods for PLC programs of industrial control systems. In: *Proceedings of the PhD Symposium at iFM'16 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM'16)*, Reykjavik University, 2016. URL: http://mit.bme.hu/~darvas/publications/PhD-iFM2016_Darvas.pdf. Extended abstract. Accepted and presented, in press.

[*c*9] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety PLC based control software. In: Erika Ábrahám and Marieke Huisman (eds.), *Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 9681, pp. 508–522. Springer, 2016. DOI: 10.1007/978-3-319-33693-0_32.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*c*10] András Vörös, Dániel Darvas, Vince Molnár, Attila Klenik, Ákos Hajdu, Attila Jámbor, Tamás Bartha, and István Majzik. PetriDotNet 1.5: Extensible Petri net editor and analyser for education and research. In: Fabrice Kordon and Daniel Moldt (eds.), *Application and Theory of Petri Nets and Concurrency*, Lecture Notes in Computer Science, vol. 9698, pp. 123–132. Springer, 2016. DOI: 10.1007/978-3-319-39086-4_9.

▷ *The implementation of the PetriDotNet framework is a joint work of the authors, based on the original work of Bertalan Szilvási. The saturation-based model checking algorithms were developed by D. Darvas and A. Jámbor, supervised by A. Vörös and T. Bartha. The design and development of bounded saturation-based algorithms are my contributions, supervised by A. Vörös and T. Bartha.*

[c11] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Generic representation of PLC programming languages for formal verification. In: *Proceedings of the 23rd PhD Mini-Symposium*, pp. 6–9. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2016. DOI: 10.5281/zenodo.51064.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[c12] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. A formal specification method for PLC-based applications. In: Lou Corvetti, Kathleen Riches, and Volker R.W. Schaa (eds.), *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pp. 907–910. JACoW, 2015. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF091.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[c13] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. PLCverif: A tool to verify PLC programs based on model checking techniques. In: Lou Corvetti, Kathleen Riches, and Volker R.W. Schaa (eds.), *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, pp. 911–914. JACoW, 2015. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF092.

▷ *The high-level ideas of the presented workflow are joint work of the authors. The detailed design and development of the tool is my contribution.*

[c14] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Víctor M. González Suárez, and Jan Olaf Blech. Modelling and formal verification of timing aspects in large PLC programs. In: Edward Boje and Xiaohua Xia (eds.), *Proceedings of the 19th IFAC World Congress*, IFAC Proceedings Volumes, vol. 47 (3), pp. 3333–3339. Elsevier, 2014. DOI: 10.3182/20140824-6-ZA-1003.01279.

▷ *The concrete time representation ("realistic approach", Section 4.1) is a joint contribution with B. Fernández Adiego. The abstract time representation (Section 4.2) is my own contribution. The refinement between the two approaches is the contribution of J.O. Blech.*

[c15] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, and Víctor M. González Suárez. Formal verification of complex properties on PLC programs. In: Erika Ábrahám and Catuscia Palamidessi (eds.), *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, vol. 8461, pp. 284–299. Springer, 2014. DOI: 10.1007/978-3-662-43613-4_18.

▷ *The high-level ideas of the verification workflow are joint work of the authors. The development and analysis of the model reduction methods (Section 3) are my own contributions.*

[c16] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. Bringing automated model checking to PLC program development – A CERN case study. In: Jean-Jacques Lesage, Jean-Marc Faure, José E. Ribiero Cury, and Bengt Lennartson (eds.), *Proceedings of the 12th International Workshop on Discrete Event Systems*, IFAC Proceedings Volumes, vol. 47 (2), pp. 394–399. Elsevier, 2014. DOI: 10.3182/20140514-3-FR-4046.00051.

▷ *The case study presented in this paper is a joint contribution based on my detailed definition and implementation of the verification workflow.*

[c17] Dániel Darvas, András Vörös, and Tamás Bartha. Efficient saturation-based bounded model checking of asynchronous systems. In: Ákos Kiss (ed.), *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*, pp. 259–273. Szeged, Hungary: University of Szeged, 2013. URL: http://petridotnet.inf.mit.bme.hu/publications/SPLST2013_

DarvasVorosBartha.pdf.

▷ *Own contribution, joint paper with M.Sc. thesis supervisors.*

[*c*18] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation based CTL model checking. In: Jaan Penjam (ed.), *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST'11*, pp. 149–160. Tallinn, Estonia: Tallinn University of Technology, Institute of Cybernetics, 2011. URL: http://petridotnet.inf.mit.bme.hu/publications/SPLST2011_VorosDarvasBartha.pdf.

▷ *Own contribution, joint paper with B.Sc. thesis supervisors.*

Local Conference and Workshop Papers

[*e*19] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Requirements towards a formal specification language for PLCs. In: *Proceedings of the 22nd PhD Mini-Symposium*, pp. 18–21. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2015. DOI: 10.5281/zenodo.14907.

▷ *Own contribution, joint paper with Ph.D. supervisors.*

[*e*20] Dániel Darvas and András Vörös. Szaturációalapú tesztbemenet-generálás színezett Petri-hálókkal [in Hungarian; Saturation-based test input generation using coloured Petri nets]. In: *Mesterpróba 2013. Konferenciakiadvány*, pp. 48–51. Budapest University of Technology and Economics, 2013. URL: http://petridotnet.inf.mit.bme.hu/publications/Mesterproba2013_Darvas.pdf.

▷ *Own contribution in frame of the R3-COP project, joint paper with M.Sc. thesis supervisor.*

[*e*21] Dániel Darvas. Szaturáció alapú korlátos modellellenőrzési technikák Petri-hálók analízisére [in Hungarian; Saturation based bounded model checking methods for the analysis of Petri nets]. In: *XVII. Fiatal Műszakiak Tudományos Ülésszaka*, pp. 83–86. Cluj Napoca, Romania: Erdélyi Múzeum-Egyesület Műszaki Tudományok Szakosztálya, 2012. URL: http://petridotnet.inf.mit.bme.hu/publications/FMTU2012_Darvas.pdf.

Technical Reports

[*r*22] Dániel Darvas, Enrique Blanco Viñuela, and István Majzik. *Syntax and semantics of PLCspecif*. Report EDMS 1523877. CERN, 2015. URL: https://edms.cern.ch/document/1523877.

▷ *Own contribution, joint report with Ph.D. supervisors.*

[*r*23] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, Jan Olaf Blech, and Víctor M. González Suárez. *Automated generation of formal models from ST control programs for verification purposes*. Internal Note CERN-ACC-NOTE-2014-0037. CERN, 2014. URL: http://cds.cern.ch/record/1708853/.

▷ *The formal definition of the IM language is my contribution. The transformation of PLC programs to IM is a joint contribution of the authors.*

[*r*24] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. *Transforming PLC programs into formal models for verification purposes*. Internal Note CERN-ACC-NOTE-2013-0040. CERN, 2013. URL: http://cds.cern.ch/record/1629275/.

▷ *The definition of the IM language is my contribution. The transformation of PLC programs to IM and the NuSMV representation of the IM are joint contributions of the authors.*

Additional Publications (Not Linked to Theses)

Journal Papers

[*j*25] Vince Molnár, András Vörös, Dániel Darvas, Tamás Bartha, and István Majzik. Component-wise incremental LTL model checking. *Formal Aspects of Computing* 28(3), 2016, pp. 345–379. DOI: 10.1007/s00165-015-0347-x. $IF_{2014} = 0.80$.

[*j*26] András Vörös, Dániel Darvas, Attila Jámbor, and Tamás Bartha. Advanced saturation-based model checking of well-formed coloured Petri nets. *Periodica Polytechnica, Electrical Engineering and Computer Science* 58(1), 2014, pp. 3–13. DOI: 10.3311/PPee.2080.

International Conference and Workshop Papers

[*c*27] Vince Molnár, Dániel Darvas, András Vörös, and Tamás Bartha. Saturation-based incremental LTL model checking with inductive proofs. In: Christel Baier and Cesare Tinelli (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9035, pp. 643–657. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_58.

[*c*28] Tamás Bartha, András Vörös, Attila Jámbor, and Dániel Darvas. Verification of an industrial safety function using coloured Petri nets and model checking. In: *Proceedings of the 14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2012)*, pp. 472–485. Hungarian Academy of Sciences, Computer and Automation Research Institute, 2012. URL: http://petridotnet.inf.mit.bme.hu/publications/MITIP2012_BarthaEtAl.pdf.

[*c*29] András Vörös, Tamás Bartha, Dániel Darvas, Tamás Szabó, Attila Jámbor, and Ákos Horváth. Parallel saturation based model checking. In: *Proceedings of the 10th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 94–101. IEEE, 2011. DOI: 10.1109/ISPDC.2011.23.

Additional Work

[*a*30] Dániel Darvas. Incremental extension of the saturation algorithm-based bounded model checking of Petri nets. Master's thesis. Budapest University of Technology and Economics, 2014. URL: http://petridotnet.inf.mit.bme.hu/publications/Diplomaterv2013_Darvas.pdf.

[*a*31] Dániel Darvas. Petri-háló alapú formális modellek analízise hatékony korlátos modellellenőrzési technikák segítségével [in Hungarian; Efficient bounded model checking techniques for Petri net based formal models]. Bachelor's thesis. Budapest University of Technology and Economics, 2011.

[*a*32] Dániel Darvas and Attila Jámbor. Komplex rendszerek modellezése és verifikációja [in Hungarian; Modeling and verification of complex systems]. Scientific Students' Association Report. 2011. URL: http://petridotnet.inf.mit.bme.hu/publications/TDK2011_DarvasJambor.pdf.

▷ *The extension of the saturation algorithms to coloured Petri nets is the contribution of A. Jámbor. The saturation-based bounded model checking algorithm is my own contribution.*

[*a*33] Dániel Darvas. Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez [in Hungarian; Implementing a saturation-based model checker of asynchronous systems]. Scientific Students' Association Report. 2010. URL: http://petridotnet.inf.mit.bme.hu/publications/OTDK2011_Darvas.pdf.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science* 126(2), 1994, pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[All70] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices* 5(7), 1970, pp. 1–19. DOI: 10.1145/390013.808479.

[Aml+05] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In: Dominique Borrione and Wolfgang Paul (eds.), *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science, vol. 3725, pp. 254–268. Springer, 2005. DOI: 10.1007/11560548_20.

[Amn+01] Tobias Amnell et al. UPPAAL – Now, next, and future. In: *Modeling and Verification of Parallel Processes*, Lecture Notes in Computer Science, vol. 2067, pp. 99–124. Springer, 2001. DOI: 10.1007/3-540-45510-8_4.

[Avi+04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1(1), 2004, pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[Bac81] Ralph-Johan R. Back. On correct refinement of programs. *Journal of Computer and System Sciences* 23(1), 1981, pp. 49–68. DOI: 10.1016/0022-0000(81)90005-2.

[Bar12] Haniel Moreira Barbosa. Formal verification of PLC programs using the B Method. Master's thesis. Federal University of Rio Grande do Norte, 2012.

[BBK12] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Arcade.PLC: A verification platform for programmable logic controllers. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 338–341. IEEE, 2012. DOI: 10.1145/2351676.2351741.

[Bec+15] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. Regression verification for programmable logic controller software. In: Michael Butler, Sylvain Conchon, and Fatiha Zaïdi (eds.), *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, vol. 9407, pp. 234–251. Springer, 2015. DOI: 10.1007/978-3-319-25423-4_15.

[Bee02] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling* 1(2), 2002, pp. 130–141. DOI: 10.1007/s10270-002-0012-8.

[Bee94] Michael von der Beeck. A comparison of Statecharts variants. In: Hans Langmaack, Willem-Paul de Roever, and Jan Vytopil (eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, vol. 863, pp. 128–148. Springer, 1994. DOI: 10.1007/3-540-58468-4_163.

[Beh+99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of B in a large project. In: Jeannette M. Wing, Jim Woodcock, and Jim Davies (eds.), *FM'99 – Formal Methods*, Lecture Notes in Computer Science, vol. 1708, pp. 369–387. Springer, 1999. DOI: 10.1007/3-540-48119-2_22.

[Bel+10] Houda Bel Mokadem, Béatrice Bérard, Vincent Gourcuff, Olivier de Smet, and Jean-Marc Roussel. Verification of a timed multitask system with UPPAAL. *IEEE Transactions on Computers Transactions on Automation Science and Engineering* 7(4), 2010, pp. 921–932. DOI: 10.1109/TASE.2010.2050199.

[BG92] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19(2), 1992, pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V.

[BG99] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1633, pp. 274–287. Springer, 1999. DOI: 10.1007/3-540-48683-6_25.

[Bha13] Sriman Kumar Bhattacharya. *Control Systems Engineering*. 3rd edition. Pearson India, 2013.

[Bia16] Sebastian Biallas. Verification of programmable logic controller code using model checking and static analysis. Ph.D. thesis. RWTH Aachen, 2016.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. In: Advances in Computers, vol. 58, pp. 117–148. Elsevier, 2003. DOI: 10.1016/S0065-2458(03)58003-2.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In: W. Rance Cleaveland (ed.), Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[BKS12] Sebastian Biallas, Stefan Kowalewski, and Bastian Schlich. Range and value-set analysis for programmable logic controllers. In: *11th International Workshop on Discrete Event Systems*, IFAC Proceedings Volumes, vol. 45 (29), pp. 378–383. IFAC, 2012. DOI: 10.3182/20121003-3-MX-4033.00060.

[Bla+11] Enrique Blanco Viñuela, Jean-Michel Beckers, Benjamin Bradu, Philippe Durand, Borja Fernández Adiego, Silvia M. Izquierdo Rosas, Alexey Merezhin, Jeronimo Ortola Vidal, Jacques Rochez, and David Willeman. UNICOS evolution: CPC version 6. In: Marie Robichon et al. (eds.), *Proceedings of the 12th International Conference on Accelerator & Large Experimental Physics Control Systems*, pp. 786–789. JACoW, 2011. URL: http://cds.cern.ch/record/1402490.

[BMP15] Sandrine Blazy, Andre Maroneze, and David Pichardie. Verified validation of program slicing. In: *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pp. 109–117. ACM, 2015. DOI: 10.1145/2676724.2693169.

[Bol15] William Bolton. *Programmable Logic Controllers*. 6th edition. Newnes, 2015.

[BS93] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal* 8(4), 1993, pp. 189–209. DOI: 10.1049/sej.1993.0025.

[Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 1992, pp. 142–170. DOI: 10.1016/0890-5401(92)90017-A.

[But02] Michael Butler. A system-based approach to the formal development of embedded controllers for a railway. *Design Automation for Embedded Systems* 6(4), 2002, pp. 355–366. DOI: 10.1023/A:1016503426126.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In: Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg (eds.), *Lectures on Concurrency and Petri Nets*, Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer, 2004. DOI: 10.1007/978-3-540-27755-2_3.

[Can+00] Géraud Canet, Sandrine Couffin, Jean-Jacques Lesage, Antoine Petit, and Philippe Schnoebelen. Towards the automatic verification of PLC programs written in instruction list. In: *IEEE International Conference on Systems, Man & Cybernetics*, vol. 4, pp. 2449–2454. IEEE, 2000. DOI: 10.1109/ICSMC.2000.884359.

[Cav+14] Roberto Cavada et al. The nuXmv symbolic model checker. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_22.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logic of Programs*, Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer, 1982. DOI: 10.1007/BFb0025774.

[CES09] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: Algorithmic verification and debugging. *Communications of the ACM* 52(11), 2009, pp. 74–84. DOI: 10.1145/1592761.1592781.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[Cha+02] Pankaj Chauhan, Edmund M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: Mark D. Aagaard and John W. O'Leary (eds.), *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, vol. 2517, pp. 33–51. Springer, 2002. DOI: 10.1007/3-540-36126-X_3.

[Che+14] Chih-Hong Cheng, Chung-Hao Huang, Harald Ruess, and Stefan Stattelmann. G4LTL-ST: Automatic generation of PLC programs. In: Armin Biere and Roderick Bloem (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 8559, pp. 541–549. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_36.

[Cia07] Gianfranco Ciardo. Data representation and efficient solution: A decision diagram approach. In: Marco Bernardo and Jane Hillston (eds.), *Formal Methods for Performance Evaluation*, Lecture Notes in Computer Science, vol. 4486, pp. 371–394. Springer, 2007. DOI: 10.1007/978-3-540-72522-0_9.

[Cim+12] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal verification and validation of ERTMS industrial railway train spacing system. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 7358, pp. 378–393. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_29.

[Cla08] Edmund M. Clarke. The birth of model checking. In: Orna Grumberg and Helmut Veith (eds.), *25 Years of Model Checking*, Lecture Notes in Computer Science, vol. 5000, pp. 1–26. Springer, 2008. DOI: 10.1007/978-3-540-69850-0_1.

[CLS00] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In: Mogens Nielsen and Dan Simpson (eds.), *Application and Theory of Petri Nets 2000*, Lecture Notes in Computer Science, vol. 1825, pp. 103–122. Springer, 2000. DOI: 10.1007/3-540-44988-4_8.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2031, pp. 328–342. Springer, 2001. DOI: 10.1007/3-540-45319-9_23.

[CLS09] Marco Colla, Tiziano Leidi, and Mario Semo. Design and implementation of industrial automation control systems: A survey. In: *7th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 570–575. 2009. DOI: 10.1109/INDIN.2009.5195866.

[CMS03] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2619, pp. 379–393. Springer, 2003. DOI: 10.1007/3-540-36577-X_27.

[CMS06] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer* 8(1), 2006, pp. 4–25. DOI: 10.1007/s10009-005-0188-7.

[CMS08] José C. Campos, José Machado, and Eurico Seabra. Property patterns for the formal verification of automated production systems. In: *Proceedings of the 17th IFAC World Congress*, IFAC Proceedings Volumes, vol. 41 (2), pp. 5107–5112. Elsevier, 2008. DOI: 10.3182/20080706-5-KR-1001.00858.

[CNQ05] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Are BDDs still alive within sequential verification? *International Journal on Software Tools for Technology Transfer* 7(2), 2005, pp. 129–142. DOI: 10.1007/s10009-004-0172-7.

[Cop+01] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In: Gérard Berry, Hubert Comon, and Alain Finkel (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2102, pp. 436–453. Springer, 2001. DOI: 10.1007/3-540-44585-4_43.

[CS02] Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In: *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, vol. 2517, pp. 256–273. Springer, 2002. DOI: 10.1007/3-540-36126-X_16.

[CS03] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In: *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2725, pp. 40–53. Springer, 2003. DOI: 10.1007/978-3-540-45069-6_4.

[CT12] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2nd edition. Morgan Kaufmann, 2012.

[CY05] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: *Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science, vol. 3725, pp. 146–161. Springer, 2005. DOI: 10.1007/11560548_13.

[CZJ12] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Ten years of saturation: A Petri net perspective. In: *Transactions on Petri Nets and Other Models of Concurrency V*, Lecture Notes in Computer Science, vol. 6900, pp. 51–95. Springer, 2012. DOI: 10.1007/978-3-642-29072-5_3.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In: *Proceedings of the 21st International Conference on Software Engineering*, pp. 411–420. ACM, 1999. DOI: 10.1145/302405.302672.

[DDd03] Alexandre David, Johann Deneux, and Julien d'Orso. *A formal semantics for UML statecharts*. Tech. rep. 2003-010. Uppsala University, 2003. URL: http://www.it.uu.se/research/reports/2003-010/.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* 19(1), 2001, pp. 45–80. DOI: 10.1023/A:1011227529550.

[Dij01] Edsger W. Dijkstra. The end of computing science? *Communications of the ACM* 44(3), 2001, p. 92. DOI: 10.1145/365181.365217.

[Din+06] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Bonnie Webber. Extracting formal specifications from natural language regulatory documents. In: Johan Bos and Alexander Koller (eds.), *Inference in Computational Semantics ICoS-5*, pp. 17–26. 2006. URL: http://anthology.aclweb.org/W/W06/W06-3902.pdf.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(7), 2008, pp. 1165–1178. DOI: 10.1109/TCAD.2008.923410.

[Dub11] Alpana Dubey. Evaluating software engineering methods in the context of automation applications. In: *9th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 585–590. IEEE, 2011. DOI: 10.1109/INDIN.2011.6034944.

[EF18] Cindy Eisner and Dana Fisman. Temporal logic made practical. In: Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith (eds.), *Handbook of Model Checking*, Springer, 2018. URL: http://www.cis.upenn.edu/~fisman/documents/EF_HBMC14.pdf. To appear.

[Fer14] Borja Fernández Adiego. Bringing automated formal verification to PLC program development. Ph.D. thesis. University of Oviedo, 2014. URL: http://cds.cern.ch/record/1983193.

[Fix08] Limor Fix. Fifteen years of formal property verification in Intel. In: Orna Grumberg and Helmut Veith (eds.), *25 Years of Model Checking*, Lecture Notes in Computer Science, vol. 5000, pp. 139–144. Springer, 2008. DOI: 10.1007/978-3-540-69850-0_8.

[FL00] Georg Frey and Lothar Litz. Formal methods in PLC programming. In: *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4, pp. 2431–2436. IEEE, 2000. DOI: 10.1109/ICSMC.2000.884356.

[FMW05] Harry Foster, Erich Maschner, and Yaron Wolfsthal. IEEE 1850 PSL: The next generation. In: *Proceedings of the Design and Verification Conference and Exhibition (DVCON)*, 2005. URL: http://212.199.43.83/papers/ieee1850psl-the_next_generation.pdf.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. 1st edition. Springer, 2000. DOI: 10.1007/978-3-662-04293-9.

[GCG10] Eugen Ioan Gergely, Laura Coroiu, and Alexandru Gacsadi. Design of safe PLC programs by using Petri nets and formal methods. In: *Recent Advances in Automation & Information – Proceedings of the 11th WSEAS International Conference on Automation & Information*, pp. 86–91. WSEAS, 2010.

[Gla+07] Ziv Glazberg, Mark Moulin, Avigail Orni, Sitvanit Ruah, and Emmanuel Zarpas. PSL: Beyond hardware verification. In: S. Ramesh and Prahladavaradan Sampath (eds.), *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pp. 245–260. Springer, 2007. DOI: 10.1007/978-1-4020-6254-4_19.

[GLK13] Giovanni Godena, Tomaž Lukman, and Gregor Kandare. A new approach to control systems software development. In: Stanko Strmčnik and Đani Juričić (eds.), *Case Studies in Control*, pp. 363–406. Springer, 2013. DOI: 10.1007/978-1-4471-5176-0_12.

[Gre94] A. Greenway. A user's perspective of programmable logic controllers (PLCs) in safety-related applications. In: Felix Redmill and Tom Anderson (eds.), *Technology and Assessment of Safety-Critical Systems*, pp. 1–20. Springer, 1994. DOI: 10.1007/978-1-4471-2082-7_1.

[GSF08] Vincent Gourcuff, Olivier de Smet, and Jean-Marc Faure. Improving large-sized PLC programs verification using abstractions. In: *Proceedings of the 17th IFAC World Congress*, IFAC Proceedings Volumes, vol. 41 (2), pp. 5101–5106. Elsevier, 2008. DOI: 10.3182/20080706-5-KR-1001.00857.

[Hal+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE* 79(9), 1991, pp. 1305–1320. DOI: 10.1109/5.97300.

[Hal05] Nicolas Halbwachs. A synchronous language at work: The story of Lustre. In: *Proceedings of the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pp. 3–11. IEEE, 2005. DOI: 10.1109/MEMCOD.2005.1487884.

[Ham05] Grégoire Hamon. A denotational semantics for Stateflow. In: *Proceedings of the 5th ACM International Conference on Embedded Software*, pp. 164–172. ACM, 2005. DOI: 10.1145/1086228.1086260.

[Har07] David Harel. Statecharts in the making: A personal account. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, 5-1–5-43. ACM, 2007. DOI: 10.1145/1238844.1238849.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8(3), 1987, pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.

[Häs13] David Hästbacka. Developing modern industrial control applications: On information models, methods and processes for distributed engineering. Ph.D. thesis. Tampere University of Technology, 2013.

[Hav+00] Klaus Havelund, Mike Lowry, Seung Joon Park, Charles Pecheur, John Penix, Willem Visser, and Jon L. White. Formal analysis of the Remote Agent before and after flight. In: C. Michael Holloway (ed.), *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, pp. 163–174. NASA, 2000.

[Hax10] Anne E. Haxthausen. *An introduction to formal methods for the development of safety-critical applications*. Tech. rep. Technical University of Denmark, 2010. URL: http://www2.imm.dtu.dk/courses/02263/F13/Files/FormalMethodsNoteTS.pdf.

[Hel16] Petra van den Helder. Verification of PLC code used at CERN. Master's thesis. Eindhoven University of Technology, 2016.

[HK99] Alexander Holt and Ewan Klein. A semantically-derived subset of English for hardware verification. In: *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*, pp. 451–456. Association for Computational Linguistics, 1999. DOI: 10.3115/1034678.1034747.

[HKB08] Markus Herrmannsdörfer, Sascha Konrad, and Brian Berenbach. Tabular notations for state machine-based specifications. *CrossTalk* 21(3), 2008, pp. 8–23.

[HLR98] Mats P.E. Heimdahl, Nancy G. Leveson, and Jon D. Reese. Experiences from specifying the TCAS II requirements using RSML. In: *Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conference*, vol. 1, pp. C43/1–C43/8. IEEE, 1998. DOI: 10.1109/DASC.1998.741499.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. URL: http://www.usingcsp.com/cspbook.pdf.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. In: Krzysztof R. Apt (ed.), *Logics and Models of Concurrent Systems*, NATO ASI, vol. 13, pp. 477–498. Springer, 1985. DOI: 10.1007/978-3-642-82453-1_17.

[HT03] David Harel and P. S. Thiagarajan. Message sequence charts. In: Luciano Lavagno, Grant Martin, and Bran Selic (eds.), *UML for Real*, pp. 77–105. Kluwer Academic Publishers, 2003. DOI: 10.1007/0-306-48738-1_4.

[Huu03] Ralf Huuck. Software verification for programmable logic controllers. Ph.D. thesis. University of Kiel, 2003.

[I1012] *IEEE Std 1012-2012 – IEEE Standard for system and software verification and validation*. IEEE, 2012.

[I13568] *ISO/IEC 13568 Information technology – Z formal specification notation – Syntax, type system and semantics*. ISO/IEC, 2002.

[I13817-1] *ISO/IEC 13817-1 Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. ISO/IEC, 1996.

[I60848] *IEC 60848 – GRAFCET specification language for sequential function charts*. IEC, 2013.

[I61131-3] *IEC 61131-3 Programmable controllers – Part 3: Programming languages*. IEC, 2013.

[I61499-1] *IEC 61499-1 Function blocks – Part 1: Architecture*. IEC, 2012.

[I61508-2] *IEC 61508-2 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems*. IEC, 2010.

[I61508-3] *IEC 61508-3 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. IEC, 2010.

[I61511-1] *IEC 61511-1 Functional safety – Safety instrumented systems for the process industry sector – Part 1: Framework, definitions, system, hardware and software requirements*. IEC, 2003.

[I61511-2] *IEC 61511-2 Functional safety – Safety instrumented systems for the process industry sector – Part 2: Guidelines for the application of IEC 61511-1:2016*. IEC, 2016.

[I62424] *IEC 62424 – Representation of process control engineering – Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools*. IEC, 2008.

[I830] *IEEE Std 830-1998 – IEEE Recommended Practice for Software Requirements Specifications*. IEEE, 1998.

[I8807] *ISO 8807 Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO, 1989.

[Int14] Intel. Intel reports record third-quarter revenue of $14.6 billion. News release. 2014. URL: https://newsroom.intel.com/news-releases/intel-reports-record-third-quarter-revenue-of-14-6-billion/.

[Jee+10] Eunkyoung Jee, Seungjae Jeon, Sungdeok Cha, Kwangyong Koh, Junbeom Yoo, Geeyong Park, and Poonghyun Seong. FBDVerifier: Interactive and visual analysis of counterexample in formal verification of function block diagram. *Journal of Research and Practice in Information Technology* 42(3), 2010, pp. 171–188. URL: http://ws.acs.org.au/jrpit/JRPIT42.3.171.pdf.

[Kai+09] Roope Kaivola et al. Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In: Ahmed Bouajjani and Oded Maler (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 5643, pp. 414–429. Springer, 2009. DOI: 10.1007/978-3-642-02658-4_32.

[Kan+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-performance language-independent model checking. In: Christel Baier and Cesare Tinelli (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_61.

[KLC06] Jin-Hyun Kim, Na Young Lee, and Jin-Young Choi. Formal specification and verification of PLC for certification. *ACM SIGBED Review* 3(4), 2006. http://www.cs.virginia.edu/sigbed/archives/2006-10/08_ITCES06_Kim_Lee_Choi.pdf.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. Bibliotheca mathematica. North-Holland, 1952.

[Kni+97] John C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Luís G. Nakano. Why are formal methods not used more widely? In: C. Michael Holloway and Kelly J. Hayhurst (eds.), *Fourth NASA Langley Formal Methods Workshop (LFM)*, pp. 1–12. 1997. URL: http://www.cs.virginia.edu/~jck/publications/lfm.97.pdf.

[Koo+06] Seo Ryong Koo, Poong Hyun Seong, Junbeom Yoo, Sung Deok Cha, Cheong Youn, and Hyun-Chul Han. NuSEE: An integrated environment of software specification and V&V for PLC based safety-critical systems. *Nuclear Engineering and Technology* 38(3), 2006, pp. 259–276.

[KSK15] Shrawan Kumar, Amitabha Sanyal, and Uday P. Khedker. Value slice: A new slicing concept for scalable property checking. In: Christel Baier and Cesare Tinelli (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 9035, pp. 101–115. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_7.

[LÅF10] Oscar Ljungkrantz, Knut Åkesson, and Martin Fabian. Practice of industrial control logic programming using library components. In: *Programmable Logic Controller*, pp. 17–32. Intech, 2010. DOI: 10.5772/7191.

[Lam+11] Thomas Lambolais, Anne-Lise Courbis, Hong-Viet Luong, and Thanh-Liem Phan. Interoperability analysis of systems. In: *Proceedings of the 18th IFAC World Congress*, IFAC Proceedings Volumes, vol. 44 (1), pp. 7879–7884. Elsevier, 2011. DOI: 10.3182/20110828-6-IT-1002.03523.

[Lam00] Axel van Lamsweerde. Formal specification: A roadmap. In: Anthony Finkelstein (ed.), *Proceedings of the Conference on The Future of Software Engineering (ICSE)*, pp. 147–159. ACM, 2000. DOI: 10.1145/336512.336546.

[Lam09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[Lev+91] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, Jon D. Reese, and Ruben Ortega. Experiences using Statecharts for a system requirements specification. In: *Proceedings of the Sixth International Workshop on Software Specification and Design*, pp. 31–41. IEEE, 1991. DOI: 10.1109/IWSSD.1991.213079.

[LHR99] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon D. Reese. Designing specification languages for process control systems: Lessons learned and steps to the future? In: Oscar Nierstrasz and Michel Lemoine (eds.), *Software Engineering – ESEC/FSE '99*, Lecture Notes in Computer Science, vol. 1687, pp. 127–146. Springer, 1999. DOI: 10.1007/3-540-48166-4_9.

[Liu+13] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. A formal semantics for complete UML state machines with communications. In: Einar Broch Johnsen and Luigia Petre (eds.), *Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 7940, pp. 331–346. Springer, 2013. DOI: 10.1007/978-3-642-38613-8_23.

[Lju+10] Oscar Ljungkrantz, Knut Åkesson, Martin Fabian, and Chengyin Yuan. A formal specification language for PLC-based control logic. In: *Proceedings of the 8th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 1067–1072. 2010. DOI: 10.1109/INDIN.2010.5549591.

[Lju11] Oscar Ljungkrantz. On formal specification and verification of function block applications in industrial control logic development. Ph.D. thesis. Chalmers University of Technology, 2011.

[LMM99] Diego Latella, István Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In: *Formal Methods for Open Object-Based Distributed Systems*, IFIP – The International Federation for Information Processing, vol. 10, pp. 331–347. Kluwer, 1999. DOI: 10.1007/978-0-387-35562-7_25.

[LNN13] Tim Lange, Martin R. Neuhäußer, and Thomas Noll. Speeding up the safety verification of programmable logic controller code. In: *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, vol. 8244, pp. 44–60. Springer, 2013. DOI: 10.1007/978-3-319-03077-7_4.

[LSP07] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems. In: *Proceedings of the 10th Brasilian Symposium on Formal Methods (SBMF)*, 2007. URL: http://rodin.cs.ncl.ac.uk/Publications/fm_sc_rs_v2.pdf.

[Luk+13] Tomaž Lukman, Giovanni Godena, Jeff Gray, Marjan Heričko, and Stanko Strmčnik. Model-driven engineering of process control software – Beyond device-centric abstractions. *Control Engineering Practice* 21(8), 2013, pp. 1078–1096. DOI: 10.1016/j.conengprac.2013.03.013.

[LYL11] Dong-Ah Lee, Junbeom Yoo, and Jang-Soo Lee. Equivalence checking between function block diagrams and C programs using HW-CBMC. In: Francesco Flammini, Sandro Bologna, and Valeria Vittorini (eds.), *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, vol. 6894, pp. 397–408. Springer, 2011. DOI: 10.1007/978-3-642-24270-0_29.

[Mar94] John J. Marciniak. *Encyclopedia of Software Engineering*. Vol. 1. John Wiley & Sons, 1994. DOI: 10.1002/0471028959.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In: C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[Meu10] Maarten G. Meulen. Verification of PLC source code using propositional logic. Master's thesis. Eindhoven University of Technology, 2010. URL: http://redesign.esi.nl/publications/falcon/meulen2010.pdf.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4), 1989, pp. 541–580. DOI: 10.1109/5.24143.

[MW99] Angelika Mader and Hanno Wupper. Timed automaton models for simple programmable logic controllers. In: *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 106–113. IEEE, 1999. DOI: 10.1109/EMRTS.1999.777456.

[NÁW15] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In: Thouraya Bouabana-Tebibel and Stuart H. Rubin (eds.), *Formalisms for Reuse and Systems Integration*, Advances in Intelligent Systems and Computing, vol. 346, pp. 55–78. Springer, 2015. DOI: 10.1007/978-3-319-16577-6_3.

[NB09] Erzsébet Németh and Tamás Bartha. Formal verification of safety functions by reinterpretation of functional block based specifications. In: *Formal Methods for Industrial Critical Systems*, Lecture Notes in Computer Science, vol. 5596, pp. 199–214. Springer, 2009. DOI: 10.1007/978-3-642-03240-0_17.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In: Deepak Kapur (ed.), *Automated Deduction – CADE-11*, Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer, 1992. DOI: 10.1007/3-540-55602-8_217.

[Ova+16] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of PLC software. *Software and Systems Modeling* 15(4), 2016, pp. 937–960. DOI: 10.1007/s10270-014-0448-7.

[Pan+16] Cheng Pang, Antti Pakonen, Igor Buzhinsky, and Valeriy Vyatkin. A study on user-friendly formal specification languages for requirements formalization. In: *14th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 676–682. IEEE, 2016.

[Par03] E. Andrew Parr. *Programmable Controllers: An Engineer's Guide*. 3rd edition. Newnes, 2003.

[Par10] David L. Parnas. Really rethinking 'formal methods'. *Computer* 43, 2010, pp. 28–34. DOI: 10.1109/MC.2010.22.

[Par81] David Park. Concurrency and automata on infinite sequences. In: Peter Deussen (ed.), *Theoretical Computer Science*, Lecture Notes in Computer Science, vol. 104, pp. 167–183. Springer, 1981. DOI: 10.1007/BFb0017309.

[Par92] David L. Parnas. *Tabular representation of relations*. Tech. rep. CRL Report No. 260. http://www.cas.mcmaster.ca/serg/papers/newtab.printer.pdf. Telecommunications Research Institute of Ontario, Communications Research Laboratory, 1992.

[PE10] Olivera Pavlović and Hans-Dieter Ehrich. Model checking PLC software written in function block diagram. In: *Proceedings of the International Conference on Software Testing, Verification and Validation*, pp. 439–448. IEEE, 2010. DOI: 10.1109/ICST.2010.10.

[Pha13] Thanh-Liem Phan. Modeling and verification techniques for incremental development of UML architectures. In: *Doctoral Symposium of the European Conference on Object-Oriented Programming*, 2013. URL: http://www.lirmm.fr/ecoop13/images/ds/8-paper-thanh%20liem%20phan.pdf.

[Pin07] Gergely Pintér. Model based program synthesis and runtime error detection for dependable embedded systems. Ph.D. thesis. Budapest University of Technology and Economics, 2007.

[Pou03] Guilhem Pouzancre. How to diagnose a modern car with a formal B model? In: Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén (eds.), *ZB 2003: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science, vol. 2651, pp. 98–100. Springer, 2003. DOI: 10.1007/3-540-44880-2_7.

[PPK07] Olivera Pavlović, Ralf Pinger, and Mail Kollmann. Automated formal verification of PLC programs written in IL. In: Bernhard Beckert (ed.), *4th International Verification Workshop (VERIFY'07)*, CEUR-WS, vol. 259, pp. 152–163. 2007. URL: http://ceur-ws.org/Vol-259/paper13.pdf.

[PRF11a] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. A formal semantics for Grafcet specifications. In: *IEEE Conference on Automation Science and Engineering (CASE)*, pp. 488–494. IEEE, 2011. DOI: 10.1109/CASE.2011.6042457.

[PRF11b] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. Translating Grafcet specifications into Mealy machines for conformance test purposes. *Control Engineering Practice* 19(9), 2011, pp. 947–957. DOI: 10.1016/j.conengprac.2010.10.001.

[PRF14] Julien Provost, Jean-Marc Roussel, and Jean-Marc Faure. Generation of single input change test sequences for conformance test of programmable logic controllers. *IEEE Transactions on Industrial Informatics* 10(3), 2014, pp. 1696–1704. DOI: 10.1109/TII.2014.2315972.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In: Mariangiola Dezani-Ciancaglini and Ugo Montanari (eds.), *International Symposium on Programming*, Lecture Notes in Computer Science, vol. 137, pp. 337–351. Springer, 1982. DOI: 10.1007/3-540-11494-7_22.

[Sad11] Jan Sadolewski. Conversion of ST control programs to ANSI C for verification purposes. *e-Informatica* 5(1), 2011, pp. 65–76. DOI: 10.2478/v10233-011-0031-3.

[Sar+08] Cleber A. Sarmento, José R. Silva, Paulo E. Miyagi, and Diolino J. Santos Filho. Modeling of programs and its verification for programmable logic controllers. In: *Proceedings of the 17th IFAC World Congress*, IFAC Proceedings Volumes, vol. 41 (2), pp. 10546–10551. Elsevier, 2008. DOI: 10.3182/20080706-5-KR-1001.01786.

[SD08] André Sülflow and Rolf Drechsler. Verification of PLC programs using formal proof techniques. In: Géza Tarnai and Eckehard Schnieder (eds.), *Formal Methods for Automation and Safety in Railway and Automotive Systems*, pp. 43–50. L'Harmattan, 2008. URL: http://www.informatik.uni-bremen.de/agra/doc/konf/08_forms_VerificationPLCPrograms.pdf.

[SF11] Doaa Soliman and Georg Frey. Verification and validation of safety applications based on PLCopen safety function blocks. *Control Engineering Practice* 19(9), 2011, pp. 929–946. DOI: 10.1016/j.conengprac.2011.01.001.

[Sie02] Siemens. *SIMATIC Statement List (STL) for S7-300 and S7-400 Programming*. A5E00171232-01. Siemens, 2002.

[Sie10] Siemens. *S7-300 Instruction List*. A5E02354744-03. Siemens, 2010. URL: http://support.industry.siemens.com/cs/document/31977679.

[Sie11] Siemens. Standards compliance according to IEC 61131-3. 2011. URL: http://support.industry.siemens.com/cs/document/50204938.

[Sie14] Siemens. *SIMATIC Industrial Software SIMATIC safety – Configuring and programming*. A5E02714440-AD. Programming and operating manual. Siemens, 2014. URL: http://support.industry.siemens.com/cs/document/54110126.

[Sie98a] Siemens. *SIMATIC Statement List (STL) for S7-300 and S7-400 Programming*. C79000-G7076-C565. Reference manual. Siemens, 1998. URL: http://support.industry.siemens.com/cs/document/18653496.

[Sie98b] Siemens. *Statement List (STL) for S7-300/S7-400*. C79000-G7076-C565-01. Reference manual. Siemens, 1998. URL: http://support.industry.siemens.com/cs/document/18653496.

[Smi85] Brian Cantwell Smith. The limits of correctness. *ACM SIGCAS Computers and Society* 15(1–3), 1985, pp. 18–26. DOI: 10.1145/379486.379512.

[So95] So-Ming So. GrafTab: An innovative requirements specification method for a PLC system. Master's thesis. The University of British Columbia, 1995. URL: https://circle.ubc.ca/bitstream/handle/2429/4059/ubc_1995-0646.pdf.

[Sou+09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In: Ana Cavalcanti and Dennis R. Dams (eds.), *FM 2009: Formal Methods*, Lecture Notes in Computer Science, vol. 5850, pp. 532–546. Springer, 2009. DOI: 10.1007/978-3-642-05089-3_34.

[SS05] Tobias Schuele and Klaus Schneider. Three-valued logic in bounded model checking. In: *Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 177–186. IEEE, 2005. DOI: 10.1109/MEMCOD.2005.1487912.

[SS11] David J. Smith and Kenneth G.L. Simpson. The meaning and context of safety integrity targets. In: *Safety Critical Systems Handbook*. Elsevier, 2011. DOI: 10.1016/B978-0-08-096781-3.10001-X.

[Šus03] Richard Šusta. Verification of PLC programs. Ph.D. thesis. Czech Technical University in Prague, 2003. URL: http://susta.cz/fel/publications/sustathesis.pdf.

[Sut08] Grégoire Sutre. Software Verification. Talk at the Summer School on Verification Technology, Systems & Applications. 2008. URL: http://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/sutre1.pdf.

[SW89] David J. Smith and Kenneth B. Wood. *Engineering Quality Software*. Springer, 1989. DOI: 10.1007/978-94-009-1121-5.

[TF11a] Kleanthis Thramboulidis and Georg Frey. An MDD process for IEC 61131-based industrial automation systems. In: Zoubir Mammeri (ed.), *Proceedings of 2011 IEEE 16th Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2011. DOI: 10.1109/ETFA.2011.6059118.

[TF11b] Kleanthis Thramboulidis and Georg Frey. Towards a model-driven IEC 61131-based development process in industrial automation. *Journal of Software Engineering and Applications* 4(4), 2011, pp. 217–226. DOI: 10.4236/jsea.2011.44024.

[TKM13] Sabine Teufl, Maged Khalil, and Dongyue Mou. *Requirements for a model-based requirements engineering tool for embedded systems: Systematic literature review and survey*. White paper. fortiss GmbH, 2013. URL: http://af3.fortiss.org/research/2013/MbRE_tool_requirements_for_embedded_systems.pdf.

[Tót09] Zsófia Tóth Heinemann. Diszkrét ipari irányítórendszerek modellezése és ellenőrzése formális módszerekkel [in Hungarian]. Master's thesis. Budapest University of Technology and Economics, 2009.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software – Concepts and Tools* 17(3), 1996, pp. 103–120.

[Ule+15] Sebastian Ulewicz, Birgit Vogel-Heuser, Mattias Ulbrich, Alexander Weigl, and Bernhard Beckert. Proving equivalence between control software variants for programmable logic controllers. In: *20th IEEE International Conference on Emerging Technologies and Factory Automation*, IEEE, 2015. DOI: 10.1109/ETFA.2015.7301603.

[UML11] *Unified Modeling Language 2.4.1 – Superstructure specification*. Object Management Group, 2011. URL: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[Uni15] Milán Unicsovics. Test generation based on state machine models. Master's thesis. Budapest University of Technology and Economics, 2015.

[Val+08] Francesco Valentini, Tomasz Ladzinski, Pierre Ninin, and Luigi Scibile. Safety testing for LHC access system. In: *Proceedings of the 11th European Particle Accelerator Conference*, pp. 532–534. EPS-AG, 2008.

[Val+13] Francesco Valentini, Timo Hakulinen, Louis Hammouti, Tomasz Ladzinski, and Pierre Ninin. Formal methodology for safety-critical systems engineering at CERN. In: *Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems*, pp. 918–921. JACoW, 2013. URL: http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/papers/tucoca04.pdf.

[Vog+14] Birgit Vogel-Heuser, Daniel Schütz, Timo Frank, and Christoph Legat. Model-driven engineering of manufacturing automation software projects – A SysML-based approach. *Mechatronics* 24(7), 2014, pp. 883–897. DOI: 10.1016/j.mechatronics.2014.05.003.

[VWK05] Birgit Vogel-Heuser, Daniel Witsch, and Uwe Katzke. Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In: *International Conference on Control and Automation (ICCA)*, vol. 2, pp. 1034–1039. IEEE, 2005. DOI: 10.1109/ICCA.2005.1528274.

[Wan+09] Rui Wang, Ming Gu, Xiaoyu Song, and Hai Wan. Formal specification and code generation of programable logic controllers. In: *14th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 102–109. IEEE, 2009. DOI: 10.1109/ICECCS.2009.41.

[Wan+13] Rui Wang, Yong Guan, Luo Liming, Xiaojuan Li, and Jie Zhang. Component-based formal modeling of PLC systems. *Journal of Applied Mathematics* 2013, 2013. DOI: 10.1155/2013/721624.

[Wei15] Alexander Weigl. Regression verification for programmable logic controller software. Master's thesis. Karlsruhe Institute of Technology, 2015. URL: https://formal.iti.kit.edu/improve/pubs/weigl_thesis.pdf.

[Wei81] Mark Weiser. Program slicing. In: *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449. IEEE, 1981.

[Wie+12] Virginie Wiels, Rémi Delmas, David Doose, Pierre-Loïc Garoche, Jacques Cazin, and Guy Durrieu. Formal verification of critical aerospace software. *AerospaceLab*, 4 2012, paper AL04–10. URL: http://www.aerospacelab-journal.org/sites/www.aerospacelab-journal.org/files/AL04-10_1.pdf.

[Woo+09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys* 41(4), 2009, 19:1–19:36. DOI: 10.1145/1592434.1592436.

[WSG07] Rui Wang, Xiaoyu Song, and Ming Gu. Modelling and verification of program logic controllers using timed automata. *IET Software* 1(4), 2007, pp. 127–131. DOI: 10.1049/iet-sen:20070009.

[YCJ08] Junbeom Yoo, Sungdeok Cha, and Eunkyoung Jee. A verification framework for FBD based software in nuclear power plants. In: *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, pp. 385–392. IEEE, 2008. DOI: 10.1109/APSEC.2008.26.

[YCJ09] Junbeom Yoo, Sungdeok Cha, and Eunkyoung Jee. Verification of PLC programs written in FBD with VIS. *Nuclear Engineering and Technology* 41(1), 2009, pp. 79–90.

[YCL09] Andy Jinqing Yu, Gianfranco Ciardo, and Gerald Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *International Journal on Software Tools for Technology Transfer* 11(2), 2009, pp. 117–131. DOI: 10.1007/s10009-009-0099-0.

[Yoo+05a] Junbeom Yoo, Sungdeok Cha, Chang Hwoi Kim, and Duck Yong Song. Synthesis of FBD-based PLC design from NuSCR formal specification. *Reliability Engineering & System Safety* 87(2), 2005, pp. 287–294. DOI: 10.1016/j.ress.2004.05.005.

[Yoo+05b] Junbeom Yoo, Taihyo Kim, Sungdeok Cha, Jang-Soo Lee, and Han Seong Son. A formal software requirements specification method for digital nuclear plant protection systems. *Journal of Systems and Software* 74(1), 2005, pp. 73–83. DOI: 10.1016/j.jss.2003.10.018.

[ZC09] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In: *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, vol. 5799, pp. 368–381. Springer, 2009. DOI: 10.1007/978-3-642-04761-9_27.

[ZGS11] Hehua Zhang, Ming Gu, and Xiaoyu Song. Edola: A domain modeling and verification language for PLC systems. In: *The Sixth International Conference on Software Engineering Advances (ICSEA 2011)*, IARIA XPS Press, 2011. URL: http://hal.inria.fr/inria-00612416.