DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université de Bretagne Occidentale

HABILITATION À DIRIGER LES RECHERCHES DE

L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Ciprian TEODOROV

G∀min∃ : Exploring the Boundary Between Executable Specification Languages and Behavior Analysis Tools

HDR présentée et soutenue à l'ENSTA Bretagne, le 03 avril 2023
Unité de recherche : Lab-STICC, UMR CNRS 6285

Rapporteurs avant soutenance :

Frédéric BONIOL	Maître de Recherche, ONERA, Toulouse, France
Stéphane DUCASSE	Directeur de recherche, INRIA Lille, France
Hans VANGHELUWE	Full Professor, University of Antwerp, Belgique

Composition du Jury :

Président :	Frank SINGHOFF	Professeur des Universités, Université de Bretagne Occidentale, France
Examinateurs :	Jeff GRAY	Full Professor, University of Alabama, USA
	Alain PLANTEC	Professeur des Universités, Université de Bretagne Occidentale, France

Invité(s) :

Emmanuel GAUDIN	Ingénieur, CEO, PragmaDEV, France

Abstract [English] The formal verification community strives to prove the correctness of a specification using formal logic and mathematical proofs. The tremendous progress in computer-aided formal verification tools, along with an ever-growing number of success stories renders these methods essential in the system designer toolbox. However, with the advent of domain-specific models and languages, many formalisms are proposed for writing dynamic system specifications, each one adapted to the specific needs of the targeted domain. A new question emerges: **How to bridge the gap** between these **domain-specific formalisms**, geared toward domain experts, and the **formal verification tools**, geared towards mathematicians? One of the answers, ubiquitous in the literature, relies on using model transformations to syntactically translate the domain-specific model to the verification model. We argue that this approach is counterproductive leading to semantic multiplication, which requires equivalence proofs that can be hard to provide and maintain.

In this dissertation, I present a new semantic-level answer developed, refined, and evaluated during the last 10 years with the help of 6 postdoctoral fellows, 8 PhD candidates, and 12 collaborative projects. This approach, named G∀min∃, promises a modular, compositional, and reusable software architecture allowing the design of a wide variety of behavior exploration tools. The core building block of this approach is a language agnostic semantic-level interface, which acts as a bridge between the dynamic semantics of a domain-specific language and the behavior analysis tools. Here we propose a formalization of the interface along with some reusable operators for creating behavior analysis tools for interactive debugging, model-checking, and runtime monitoring. Besides reviewing almost a decade of fruitful research, this document allows me to introduce some new research directions, which hopefully will ease the burden of creating novel specification-design environments and render the design process more productive.

Résumé [French] La communauté de la vérification formelle s'efforce de prouver la conformité d'une spécification par l'utilisation de la logique formelle et de preuves mathématiques. Les progrès considérables réalisés dans les outils de vérification formelle assistée par ordinateur, ainsi que le nombre croissant de réussites, rendent ces méthodes essentielles dans la boîte à outils des concepteurs de systèmes. Cependant, avec l'avènement des modèles et des langages spécifiques à un domaine, un grand nombre de formalismes ont été proposés pour écrire des spécifications de systèmes dynamiques, chacun étant adapté aux besoins spécifiques du domaine ciblé. Une nouvelle question émerge : Comment combler le fossé entre ces formalismes spécifiques au domaine, orientés vers les experts du domaine et les outils de vérification formelle, orientés vers les mathématiciens ? Une des réponses, omniprésente dans la littérature, repose sur l'utilisation de transformations de modèles pour traduire syntaxiquement le modèle spécifique au domaine vers le modèle de vérification. Nous soutenons que cette approche est contre-productive et conduit à une multiplication sémantique, qui nécessite des preuves d'équivalence qui peuvent être difficiles à fournir et à maintenir.

Dans ce manuscrit, je présente une nouvelle réponse au niveau sémantique développée, raffinée et évaluée au cours des 10 dernières années avec l'aide de 6 ingénieurs postdoctoraux, 8 candidats au doctorat et 12 projets collaboratifs. Cette approche, nommée G∀min∃, promet une architecture logicielle modulaire, compositionnelle et réutilisable permettant la conception d'une grande variété d'outils d'exploration du comportement. La brique de base de cette approche est une interface de niveau sémantique agnostique au langage, qui agit comme un pont entre la sémantique dynamique d'un langage spécifique au domaine et les outils d'analyse du comportement. Nous proposons ici une formalisation de l'interface ainsi que quelques opérateurs réutilisables pour la création d'outils d'analyse du comportement pour le débogage interactif, le contrôle de modèle et la surveillance de l'exécution. En plus de passer en revue près d'une décennie de recherches fructueuses, ce document me permet de présenter quelques nouvelles directions de recherche, qui, nous l'espérons, allégeront le poids de la création de nouveaux environnements de conception de spécifications, et rendront le processus de conception plus productif.

Foreword

The "Habilitation à diriger des recherches" is the highest French academic degree, which «*sanctions the recognition of the high scientific level of the candidate, of the original character of his scientific approach, of his ability to master a research strategy in a broad scientific field, and of his ability to supervise young researchers*»[59]. The HDR manuscript shows the scientific maturity of the candidate through an overview of his/her research activities. The document is a snapshot-like presentation of the author's works illustrating a clear vision of the subject of study, along with a scientific methodology, which ultimately leads to promising research perspectives. In other words, an HDR manuscript sketches the destination, introduces the starting point, and discusses the route taken emphasizing the new research opportunities discovered on the way. The main consequence is that the HDR manuscript is not a textbook nor a monograph, which needs to be more polished and self-contained.

This document is the manuscript of my HDR. Here, I overview almost a decade of research at the crossroads of three fields: computer science (formal specification & verification), software engineering (executable language engineering), and computer engineering (execution platform engineering). In general, my research strategy follows the design science paradigm [60]. However, the presentation is organized in a more top-down fashion, which emphasizes the lessons learned (the partial conclusions) before describing some of the experiments and intermediate results (contributions).

This document is intended mainly for the referees of my HDR, who must evaluate my research activities. This document also introduces a problem that plagues numerous practitioners, the lack of adequate tools for understanding executable systems. Moreover, it gives a brief overview of the state of the art in this field. Finally, this document would be also of interest to both scholars and practitioners interested in the design of monitoring tools for executable languages, specification language engineering, diagnosis of model dynamics, and modular software development. From this perspective, this manuscript is an optimistic account showing that sometimes following a problem-driven approach can lead to the discovery of simple and elegant software engineering principles.

Contents

Chapter 1

Introduction

Contents

Designing dynamical systems is hard due to the nonlinear nature of behavior composition operators. To cope with the complexity the designer relies on abstractions, sometimes called models or specifications, that simplify the problem by either ignoring irrelevant details and/or by generalizing to a set of behaviors. The set of behaviors described by a **correct specification** includes only the desirable behaviors of the system. However, guaranteeing the correctness of a specification is not trivial, requiring behavior exploration tools that help initially to increase the confidence that the specification matches the desired behaviors, and later show that the system behavior is within the bounds of the specification. The formal verification community strives to prove the correctness of a specification using formal logic and mathematical proofs. The tremendous progress in computer-aided formal verification tools, along with an ever-growing number of success stories renders these methods essential in the system designer toolbox. However, with the advent of domain-specific models and languages, many formalisms are proposed for writing dynamic system specifications, each one adapted to the specific needs of the targeted domain. A new question emerges: **How to bridge the gap** between these **domain-specific formalisms**, geared toward domain experts, and the **formal verification tools**, geared towards mathematicians? One of the answers, ubiquitous in the literature, relies on using model transformations to syntactically translate the domain-specific model to the verification model. We argue that this approach is counterproductive leading to semantic multiplication, which requires equivalence proofs that can be hard to provide and maintain. In this dissertation we present a new semantic-level answer

that promises a modular, compositional, and reusable software architecture allowing the design of a wide variety of behavior exploration tools. The core building block of this approach is a language agnostic semantic-level interface, which acts as a bridge between the dynamic semantics of a domain-specific language and the behavior analysis tools. This document proposes a formalization of the interface along with some reusable operators for creating behavior analysis tools for interactive debugging, model-checking, and runtime monitoring. This approach was developed and evaluated through numerous realistic studies, which showed that *a)* it is possible to use the same semantics for both formal verification, by model-checking, and bare-metal execution (deployment). *b)* the approach allows reusing, without modification, the safety properties used for verification (during the design phase) for runtime monitoring (during embedded deployment). *c)* the SLI fosters reuse, allowing for instance the reuse of a UML model interpreter for both model and property execution. *d)* the constraints imposed by our proposition are in adequation with the industry practices, which is expected to ease the adoption of the approach. *e)* the approach enables early analysis of incomplete semantic implementations and/or partial models *f)* the fine-grained modularity eases the design of new diagnosis setups, reduced multiverse debugging. Moreover, in the context of model-checking, the isolation between the model semantics and the verification algorithms eased the design of new formal verification techniques, amongst which the FPGA-accelerated swarm verification promises more scalable and orders of magnitude faster verification engines. Furthermore, the formalization presented here served as a basis for the development of 3 significant open-source research prototypes, which have been used in both academic and industrial settings. Amongst these we emphasize the OBP2 model-checking kernel, which has been integrated in two commercial products for verifying BPMN and SDL models.

This chapter summarizes some of our research activities. After a presentation of the general context, in section 1.1, the main objectives and challenges addressed are introduced in section 1.2. Once the scope of our research efforts is defined, in section 1.3 we review our principal scientific contributions. Section 1.4 briefly discusses the research methodology we follow. Scientific research is by nature collaborative, sadly it is impossible to acknowledge all sources of inspiration. Nevertheless, in section 1.5 we overview the most fruitful collaborations, and in section 1.6 we list the research grants, industrial contracts, and collaborative projects that allowed us to conduct our research activities.

1.1 Context

Model-Driven Engineering (MDE) promises to reduce the accidental complexity in the development of complex software-centered systems [61]. The root cause of this complexity is traced to the gap between the problem-specific concepts manipulated by the domain experts and the abstraction level provided by general-purpose programming languages. MDE approaches address this problem by introducing domain-specific models along with engineering techniques and tools which help to separate and contextualize the efforts. For modeling dynamic behaviors, these domain-specific models become executable languages, sometimes called executable domain-specific languages (xDSL). In some instances (eg fUML[62]), the dynamic semantics of xDSL shares some features with the general-purpose programming languages. In other instances, the dynamic semantics is radically different [63].

Amongst xDSL one can distinguish the subclass of **executable-specification (ES) languages**, as the set of domain-specific languages which strive *to capture* the dynamic behaviors of complex systems so that they can *be studied* in captivity. The need to model and understand the dynamics of physical processes led to the invention of calculus by Newton and Leibniz, which can be seen as a subclass of the ES language family. More recently the history of the ES language family

became intertwined with the evolution of computer science. Reasoning on the dynamics of computer systems helped, amongst others, to identify other useful subclasses, such as finite-state automata, Petri nets, and lambda-calculus. The study of these formalisms is one of the pillars of theoretical computer science, a discipline that aims to understand the nature of computation.

The push toward formal. Tremendous development in this field led to the development of numerous methodologies and tools, which help practitioners to **think above the code** [64] and sometimes even derive formal mathematical statements as a result. The theoretical advances and the numerous technological breakthroughs in this area led to a push towards wider adoption of *formal reasoning and modeling*. In this trend, "The Science of Deep Specification" project[1], for instance, focuses on the specification and verification of full functional correctness of software and hardware [65]. The International Council on Systems Engineering emphasized the need for adopting formal modeling techniques for all steps of the development process (specification, analysis, design, verification) [66]. The widespread adoption of formal modeling promises the production of more reliable systems at a lower cost, using mathematics. With time this direction might even ease the pain of today's systems engineers still struggling to move from a document-centric to a model-based approach while generalizing the outdated prediction and control metal mindset [67].

Reflexivity and Agility With Lisp, John McCarthy [68, 69], showed that data and code can mix and that both can dynamically co-evolve. The Smalltalk language added a biological metaphor to computing. That of *"protected universal cells"*, today largely known as objects, *"interacting only through messages that could mimic any desired behavior"* [70]. This reification of the conceptual artifacts manipulated by programmers, allowing the isolation of concerns through inter-connected objects, nourished the development of the MDE field. Live programming, the capacity of editing the code of a running program, was also enabled by these initial developments and fed into the need for agility of software developers. This led to the development of systems like the lively kernel [71]. Some recent works introduce the concepts of live modeling in the MDE field [72]. Software is intrinsically dynamic, understanding dynamical systems is hard, thus we can argue that live-programming leverages some natural human desires (such as interaction, socializing, learning, and self-expression), to ease the act of programming. This approach resembles gamification [73], a strategic attempt to enhance the programming activity in order to keep the programmers engaged and motivated.

Language monitoring [74] is the process of observing the execution of a computer program expressed in a given programming language. In the following, the tools that enable this process will be referred to as language monitors, or simply monitors[2]. Reflexive programming languages like Lisp [69] and Smalltalk [75] streamline the gamification process by providing access to the language implementation from the language itself. The program execution can be manipulated from the program itself, which allows the development of powerful execution analysis tools. Executable specification languages are not necessarily reflexive, and external components are needed for manipulating the execution, provided that the implementation of the semantics exposes the necessary interface (Semantic Language Interface(SLI) in Figure 1.1). In the following, we will generically refer to the analysis tools as *execution management* components, language monitors, or, simply, monitors.

The computer as an *execution platform* for analyzing the dynamics of an executable-specification, offers an ideal gamification environment. The platform is generic, allowing the simulation of a

[1] https://deepspec.org/

[2] The notion of monitor here is more general than the notion of monitor in the context of *runtime monitoring*, which focuses on observing the execution only during the deployment phase.

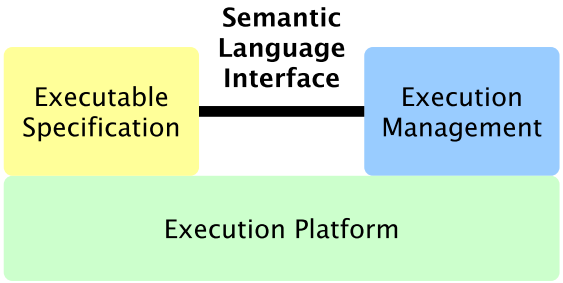


Figure 1.1: Research Context

wide range of dynamic semantics, and flexible, allowing the creation of arbitrarily complex analysis tools. However, when capturing the dynamics of the computer system itself in an ES, careful isolation is needed between the analysis platform and the deployment platform in order to guarantee the soundness of the analysis results. Moreover, the executable specification language and the underlying tools need to allow the integration in the specification of constraints emanating from the execution platform used for deployment.

1.2 Objectives and Challenges

The push toward formal rigor can sometimes be at odds with the dynamicity and agility required for understanding complex executable specifications. Most importantly, a formal executable specification environment should ease the diagnosis process by allowing the creation of generic or domain-specific execution monitors, which offer different perspectives on the underlying behavior. The research efforts presented in this manuscript are focused on understanding the connection between *Executable Specification* languages and their *Execution Management* in the context of varied *Execution Platforms*, see Figure 1.1. These efforts are situated at the confluence of multiple disciplines such as language design and implementation, software reuse, modeling and metamodeling, software diagnostic and formal verification, embedded systems, and reconfigurable circuit design.

1.2.1 Specifying dynamical systems is hard

Designing dynamical systems is inherently hard due to the emergence of potentially unwanted behaviors during composition. Writing specifications is hard since no amount of automation will allow bridging the gap between the dream (the client's desire to acquire a given capability) and reality (the system providing exactly the desired capability). The practitioners describing the capabilities of a future dynamic system face both these challenges. Verification techniques, including formal verification, help but only if the desired capability is already specified (informally or formally). Only then we can verify that a given implementation satisfies the specification. **Objective** *Designing Executable Specification should be as fun and productive as programming in a dynamic language, such as Smalltalk, and the result should be as sound as formal mathematics.* To achieve this goal the specification designer needs tools (IDEs, debuggers, profilers, test frameworks, model-checkers, provers) initially only to increase its confidence that the (potentially partial) specification faithfully matches the desired capability, and then and only then to prove that a system implements (refines) the specification.

The creation of dedicated monitoring tools for each new language is very costly in terms of

development effort, the resulting tools are not reusable even for closely related projects, and such targeted effort closes the tool exploration axis of the whole design-space exploration problem by directly providing *supposedly* optimized solutions.

In the language design community, the design-space exploration problem is typically studied from the perspective of the adequacy between the domain and the language. In this context, there have been a lot of research efforts to tune the language to the domain needs. The monitor optimization (tool optimization) is viewed as an independent problem, and many improvements in terms of algorithm complexity, scalability, and flexibility were achieved by each monitor used during the design and verification phases. In this manuscript, we argue that there is a stringent need for adding a third dimension to the design-space exploration focused on tool design and optimization. This three-dimensional view (domain/language/tool) adds two important perspectives to the DSE problem, namely the adequacy tool/language and the adequacy tool/domain. In the context of this manuscript, **the focus is on tool/language adequacy**, the tool/domain adequacy is treated as a secondary issue. The importance of this new perspective comes mainly from the need of tool reuse (to reduce the development costs) and from the need for unbiased evaluation of different technological frameworks at the architectural level (to objectively compare different solutions).

Lack of tools for temporal logic specifications Temporal logic emerged in the 80 as a good specification language for software verification. Today, 40 years later we do not have any temporal languages offering basic tools such as IDE, debugger, and profiler. Numerous attempts have been made, and the literature is rich with contributions in this direction. Nevertheless, the formal language design teams, understandably, focus their efforts on formal language design and proofs and not on "simple tools", such as debuggers. The question is how can the software community help them? During the last years, tremendous progress on domain-specific language design greatly improved our understanding of their advantages and limitations. From the language tooling perspective, these research efforts culminate with the introduction of the Language Server Protocol [3] (LSP), which decouples the integrated development environments (IDE) from the programming language implementation, allowing their independent evolution. The main breakthrough of LSP was to focus on an open API design that enabled usage well beyond the initial target audience. Today interactive theorem provers such as Coq, Lean, and Agda, all recognize the importance of opening to wider audiences (outside emacs). This reduced barrier to entry allows transposing the IDE capacities from one language to another. This is even more important for highly sophisticated languages that go beyond traditional programming.

Language-agnostic Debugging tools [76, 77, 78, 79] and the debugger server protocol (DSP) transposed this API-based design toward the analysis of the model dynamics. While more tuned towards programming language debugging, the DSP found an audience in the formal specification community. Today TLA+ is extending the already existing implementation of the LSP protocol to DSP in an incremental fashion. The VDM community also recognizes the importance of a specification-oriented language server protocol [80].

Gemoc project [76] pushed for language-independent omniscient debugging, but with a larger target audience of executable domain-specific languages. Object-centric moldable debugging[77, 78] pushed the limits by looking at debugging not as a language-specific tool but as an "application-specific" requirement for increased productivity. Moreover, Object-centric debugging works in practice, the real-world applicability being showcased by its integration in the Pharo Smalltalk IDE.

Nevertheless, besides debugging, less significance was given to the reuse of other *behavioral*

[3] https://microsoft.github.io/language-server-protocol

exploration tools (such as profilers, model-checkers), which hinders the shared improvement over multiple languages. **Objective** *Characterize the toolkit needed for studying the large number of behaviors subsumed by Executable Specifications.* The purpose of an executable specification is to describe a family of behaviors. In this context, typical program analysis tools do not scale well. For instance, multiverse debugging[81] was proposed as an extension of omniscient debugging for non-deterministic executions. The authors recognize, however, that the approach does not scale due to the state-space explosion problem. Recently, our proposition in [82] renders the approach practical through user-defined reductions. Nevertheless, further research is needed to understand the interplay between different abstraction strategies, which furthermore need to be easy to apply to existing language semantics - from an engineering perspective. Moreover, Executable Specifications debugging might require additional specific functionalities, for instance, more expressive breakpoints.

Focus on the dialog, not on the subject language. One of the core problems with language-specific tool development is that the focus is on what language features are needed to implement the tool and not what are the API-level requirements of the tool (what the tool needs to know and when - instead of what the language should have so that the tool works). Our conjecture is that switching to an API-level requirement mindset frees the tool developer from thinking in terms of subject language features to a contract-oriented mindset focused on the tools. If (one day) the language implements the API (no matter how) the tool will be available.

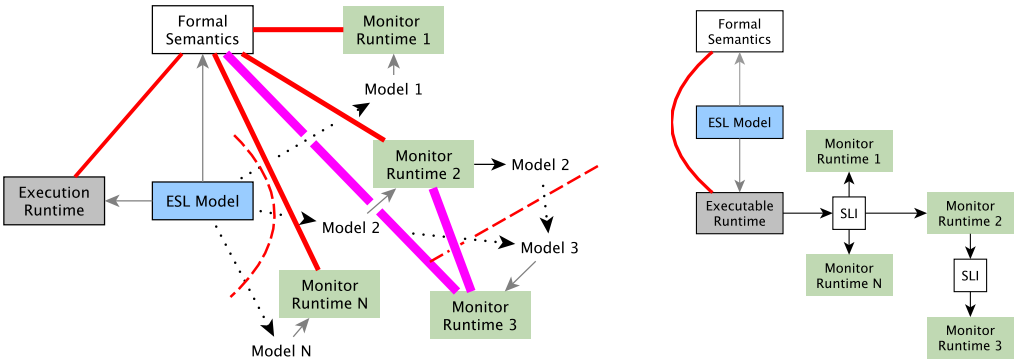
One of the hardest problems, in the context of ES monitoring, is **identifying the *language needed* for creating a semantic-level dialog between the ES language semantics and the monitoring tools, viewed themselves as dependent semantics**.

1.2.2 Too many semantics

Another problem is the extensive use of denotational semantics and of its closely related cousin (in the context of defining the semantics of an executable language) model transformations. The theoretical computer science community agrees that the best way to formalize something is by defining its meaning with mathematics. Thus, from this perspective, it is natural to define the semantics of a language, including an ES language, by mapping it to its mathematical meaning (*Formal Semantics* in Figure 1.2a). For practical purposes, the practitioners then implement the language by mapping it to an execution platform (which, by the way, is not necessarily formally defined) to get an *execution runtime* (Figure 1.2a). Suddenly, we have two semantics, the mathematical one and the practical one, which should be proved equivalent. Sadly, under time-to-market constraints, this proof is not always provided.

The matters get even worse, at a later stage when the language users want to formally verify their software, and they do not have the verification tools available along with the subject-language implementation. Here we conjecture that much of the software community is at this stage. Looking for a solution for software verification, the practitioner encounters a large offering of verification tools, most of which come with their own specification language, typically different from the subject language of interest. In Figure 1.2a this situation is illustrated by the 4 *Monitor Runtimes*, each one corresponding to a verification tool.

If we have a program (a model) written in the subject language, and a verification system based around its verification language, the natural question that arises is: "How to carry the subject-language program to the verification language ?". To address this simple practical mismatch, an engineer (or a team) could migrate each program of interest to the verification language. However, this approach is time-consuming and the manual encoding itself might introduce bugs. A conceptually better approach, backed by the software and model-driven engineering com-



(a) Semantics duplication induced by model transformations.

(b) Monitoring through a semantics language interface.

Figure 1.2: Two approaches to the language monitoring problem. The dotted arrows represent model transformations. The Dashed red lines represent the semantic gaps induced by transformations. The thick red lines represent the equivalences that need to be proved, and maintained.

munities, is to create a software transformation that can translate the source language syntax to the verification language syntax, while preserving the semantics (the dotted lines in Figure 1.2a). These transformations lie at the core of model-driven engineering, and numerous tools have been developed over the years, in 2019 a survey published in the Software & System Modeling journal identified not less that 60 model transformation tools [83]. In our context, however, defining a model transformation can be seen as writing another denotational semantics for the subject language, which already had one. However, as before, if we have two semantics, they should be proved equivalent. Testing and debugging can only increase the confidence in our model transformations, and numerous research efforts have been focused on this problem [84]. Nevertheless, the lack of semantic equivalence proof leads to doubt during the diagnosis process. Is the program faulty, is the subject-language implementation buggy, is the verification tool incorect or maybe one of the transformations was not perfect? This leads to difficulties during the diagnosis process. This problem becomes even more acute if multiple analysis tools need to be used. The number of transformations increases, as well as the probability of introducing errors in the toolflow. To make matters worse, each language-to-language transformation introduces a conceptual gap that disconnects the domain expert from its domain, which further increases frustration.

While denotational semantics and model transformation are exceptional tools, the problem is multiplying the number of semantics without really proving that they are effectively related (equivalent, or abstractions). The tremendous development of proof assistants renders these proofs almost attainable. But writing dependently typed programs (proofs) can be seen as hard for the typical programmer, especially if it must relate two real-world semantics, which tend to be less pure than their academic counterparts. Consider, for example, two semantics implemented in C, maybe with inline assembly for performance reasons. Furthermore, the composition of language monitoring tools, as illustrated in Figure 1.2a by the connection between the *Monitor Runtime 2* and the *Monitor Runtime 3*, tends to render these proofs even more difficult due to the need to related multiple semantics (the two thick pink lines in Figure 1.2a).

Another approach, inspired by a physical world metaphor, is possible. Imagine a person going to multiple doctors, the person is not transformed, simply the doctors look at the person from different angles. Sometimes, it is even beneficial to visit multiple physicians in series, each one enriching the view for the next, through diagnostic letters. Illustrated in Figure 1.2b, the

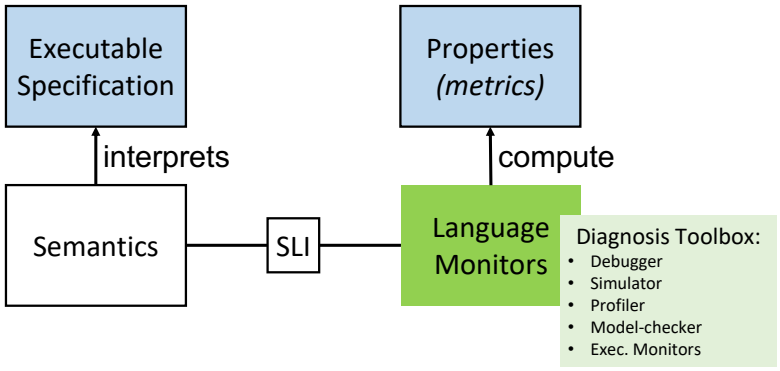


Figure 1.3: The Semantic Language Interface (SLI), a bridge between the executable specification semantics and the Language Monitors.

language runtime can let the tools look at it from different perspectives. *Monitor Runtime 1* can inspect and interact with the execution differently than *Monitor Runtime 2* or *Monitor Runtime N*. Furthermore, *Monitor Runtime 3* can look at the execution of the *ESL Model* enriched with the annotations of the *Monitor Runtime 2*. More importantly for us, this approach does not demultiplies the number of semantics but only requires collaboration, and coordination between the language designers and the tool providers.

Finally, in this context, the biggest challenge is identifying a generic, and elegant bridge between the semantics and the tools that imposes a minimal set of constraints so that it can be easily rolled into production by practitioners.

1.3 Selected Contributions

Kishon et al. in 1991 [74] identified the generic concept of *execution monitoring*, as the process of observing and controlling the execution of a program. This perspective conceptually unifies the set of tools used for manipulating and understanding the temporal evolution of a program. Furthermore, it abstracts away the complexity of the language semantics while requiring the ability to introspect and sometimes control the observable behavior exposed. Equipped with this understanding one of our initial contributions was to define the requirements for execution monitoring[11], which clearly set the **language monitoring problem** as the central problem addressed by our research activities.

To clearly set the scope of our vision, Figure 1.3 shows the *Semantic Language Interface*(SLI) as a bridge between the *Semantics* of *Executable Specification* languages and the *Properties* or metrics that are compiled from the behaviors through the *Language Monitors*, which are seen as dependent semantics. Understanding the detailed requirements imposed on the SLI bridge between the ES language semantics and the monitoring tools was the core driver that underlies most of our research contributions.

Our contributions are spread along three axes:

- **[Axis 1 – G∀min∃: The Formalization of the Semantic Language Interface]:** the design of a **modular architecture** for the **language monitoring problem**, a direction that explores the services of the SLI mainly from the perspective of the *language monitors*. Furthermore, we strive to identify generic reusable operators that ease the design of specialized language monitors[11, 1, 2, 82];

- **[Axis 2 – The Way to Modular Executable Specification Monitors]:** the **instantiation** and the **evaluation** of the G∀min∃ approach in practice through **significant research artifacts** and **industrial experiences**, in this context we focus on the instantiation of our conceptual approach in representative frameworks geared towards real-world specification languages from the academia and industry, such as TLA+[85], AEFD[86], Pimca[12, 13], UML Statechart[14, 15, 16, 17, 1, 2], BPMN[18, 19, 20];
- **[Axis 3 – New Formal Verification Techniques]:** the design of verification techniques independent-of and uncluttered-by the choice of the executable specification language. This axis is focused principally on the scalability of model-checking in practice at the model [3, 4, 21], algorithmic [5, 6, 3, 7] and platform levels [22, 23, 24].

1.3.1 Axis 1 – G∀min∃: The Formalization of the Semantic Language Interface

Our first contribution axis, which is the subject of chapter 2, focuses on the formalization of the SLI interface, an effort named G∀min∃ in the following. This formalization aims to crystallize our current understanding of the subject. G∀min∃ SLI exposes the underlying language semantics through an intensional transition system similar to the Plotkin-style operational semantics[87] extended with the functionalities needed for execution monitoring. Based on this SLI formalization we already defined some generic operators, which streamline the link creation between the semantics and the runtime environment. The power of this approach is illustrated in chapter 2 by formalizing a multiverse debugger monitor, as a composite dependent semantics integrating a model-checker monitor for finding breakpoints in ES non-deterministic execution[82]. Furthermore, we formalize multiple language monitoring setups ranging from embedded execution to complex model-checking strategies that integrate process scheduling as semantic filters over the subject language semantics [1, 2].

1.3.2 Axis 2 – The Way to Modular Executable Specification Monitors

Besides our individual contributions in peer-reviewed conferences and journals that are cited in this manuscript, from the language design perspective, four significant experiences shed some light on the realizability of our vision.

1. The design and development of the OBP2 language-agnostic requirement verification environment, our main research vehicle, shows that the proposed approach is feasible. OBP2 instantiates our conceptual architecture (described in chapter 2) offering an extensible framework that mixes omniscient and multiverse debugging with model-checking. In terms of subject language support, OBP2 currently allows the verification of executable specifications in Fiacre[88], TLA+[85], AEFD[86], EMI-UML[1], AnimUML[16], BPMN[89]. Moreover, the OBP2 implementation, done in Java, does not constrain the metalanguage used for implementing the subject-language semantics. This feature freed the toolbox from the language barrier problem, allowing the seamless integration with language runtimes implemented in bare-metal executables (for EMI-UML), javascript (for AnimULM), python (for PragmaDEV PROCESS).

> In November 2020, the open-source Hub of Pôle Systematic Paris-Region has nominated OBP to the price ”Coup de cœur académique” (the 3 other nominees were : Hardware Locality - INRIA Bordeaux ; Why3 - Equipe projet Toccata (ex ProVal) of Inria Saclay-Île-de-France, LRI Univ Paris-Saclay, CNRS; Scikit-learn - INRIA Paris-Saclay).

Currently the **research prototype: OBP2** is available as an open-source contribution at http://www.obpcdl.org.

2. Moreover, the OBP2 model-checking language monitor was integrated and is **distributed with the commercial tool PROCESS developed by PragmaDEV** [90, 19, 20] and released November 13th 2020. More recently (June 14th 2022), **PragmaDEV announced the integration of the OBP2 model-checker with Studio V6.0**, their flagship commercial product focused on the design, and verification of SDL specifications [91]. This shows that the constraints that our approach imposes on the subject-language implementation are reasonable and compatible with industrial practices.

3. Exploring the reuse of the executable semantics, as an alternative to model-transformations, for both the real execution and the verification of ES languages led us to design the **research prototype: EMI-UML** a bare-metal embedded model interpreter for UML. On the way we showed that we can unify LTL verification and embedded execution [14], we discovered that the same observer automata used for verification can be used as runtime monitors, without needed model transformations[15] and we showed that, with minimal modifications, an UML Statechart execution engine can be used to encode both the model and the properties (safety and liveness) for model-checking[1].

4. The **research prototype: Animuml**, a lively environment[71] for the debugging and the formal verification of partial UML specifications, available at http://animuml.obpcdl.org. [16, 17]. Recently in [82], we have showcased through AnimUML that user-defined sub-approximations during breakpoint lookup, are a breakthrough feature that renders the multiverse debugger monitor scalable on arbitrary models.

1.3.3 Axis 3 – New Formal Verification Techniques

The G∀min∃ SLI isolates the semantics from the execution monitors, enabling their independent evolution. Thus, it is natural to question the impact such an isolation can have on the evolution of the monitors. On this axis, we have been interested mainly in the *model-checking monitor*. Liberated from the language dependency, we have contributions focused on the scalability of explicit-state model-checking: *a)* at *the model level*, we have proposed the use of verification-guides [21] to systematically decompose the verification problem and to allow for partially-bounded verification procedures; *b)* at *the algorithmic level*, we have invented PastFree[ze] [3] a new safety verification algorithm, which by exploiting the acyclic nature of verification guides drastically improves the scalability of the partially-bounded verification procedures; *c)* at the *execution platform level*, we have proposed the first modular hardware model-checker that offers a continuum of algorithms ranging from partial to exhaustive verification [22]. Furthermore, by exploiting both the algorithmic variability and the fine-grained parallelism of reconfigurable architecture, we have discovered verification procedures almost an order of magnitude faster than the state-of-the-art [23]. These discoveries led us to the first hardware verification procedure covering both safety and liveness properties that achieves and average speedup of 4875X over software [24], a real breakthrough in the field. The use of the G∀min∃ SLI significantly decreased the efforts needed for the integration of these contributions into research prototypes. The model and algorithmic level contributions have been integrated into the OBP2 framework. The hardware verification cores have been designed as language agnostic IPs, which can be instantiated with multiple SLI-based semantics.

1.4 Research Strategy

Design science [60] is at the core of our research methodology. In the following paragraphs, we first resume our global strategy from the design science perspective. Then we address the local opportunism that arises during a constructive problem-driven approach.

A strong motivation comes from our industrial experience where we felt that the lack of tools for manipulating the dynamics of our executable specifications (heterogeneous circuit specifications, at the time) significantly hindered our productivity. Later during our postdoctoral fellowship, we were confronted with yet another problem, the demultiplication of intermediate languages needed to ease the transformation between executable specifications. In this case, the problem was even more acute, some intermediate languages lacked execution analysis tools. But even where they were present, the semantic gap between the initial specification and the one produced rendered the diagnosis process very difficult (if not impossible). **The need for solutions for these practical problems** led us to pursue the research directions discussed in this manuscript.

Methodologically our **research objectives** focus on designing *solution concepts* that ease the manipulation of executable specifications. These prescribed solutions should directly contribute to improve the state of the art.

Following design science, we mainly rely on **abductive reasoning to initiate the creative process** based on our observations. Make no mistake, these problems are not new, they are probably as old as the computer science discipline itself. Nevertheless, starting from our observations and previous experiences, we conjecture that the potential causes are the excessive use of model transformations (to create maps between semantics) and the tight link between the semantics and the algorithms that manipulate them as data (the execution monitors: debuggers, model-checkers).

To validate our hypothesis, we rely on a constructive research method. Design science research focuses on producing new knowledge while consolidating our understanding in a research artifact. This method is highly beneficial in the context of software engineering enabling iterative, almost agile, discovery. However, careful consideration is needed at the beginning of each study to assess the adequacy of existing solutions concepts as a vehicle supporting the new study. This research method led to the design of research prototypes.

Definition, decomposition, search and integration are the core steps of problem-solving. The process starts by defining a problem, which is then decomposed into smaller challenges. For each challenge, a search process starts to look for a solution. Once all subproblems are solved, the integration process connects the solutions to obtain a result for the initial problem. This process can be (and in practice it is) applied recursively during the search phase, which generates a tree decomposition of the problem. For complex problems, some of the nodes of the tree pose real scientific challenges becoming research problems. From a scientific research perspective, these nodes should be unraveled, and doing so might seem just opportunism (if we regard the research problems without considering the tree they belong to). Sadly, sometimes the tree is not explicitly present, furthermore, the scientific community does have more incentive to unravel new nodes than to participate in the integration phase, which is costly and seen as the prerogative of industrial practitioners. However, when looking at the problem decomposition tree from the perspective of software engineering, and at the results of the search phase as software artifacts, then the integration problem becomes itself a node in its own higher-level problem tree. In the context of our research, we are sometimes in such a situation, forced to unravel nodes in different problem trees while working on integration problems in other higher-level trees. Furthermore, it sometimes can be difficult to find the roots of the trees we are unraveling if they are not explicitly laid down. Even worse, we can find ourselves unraveling the nodes of a tree, for which the root suddenly disappears due to seemingly unrelated scientific breakthroughs. In such situations, we should remind ourselves that our role in society is to take those risks that the industry

cannot afford to take, with the sole purpose of expanding our understanding. This problem decomposition perspective can be even liberating as the requirements for scientific research are just: **be curious, explore, expand our understanding, and share the insights**[4].

1.5 Supervision

In 1159 John of Salisbury wrote in his book Metalogicon: *"We are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours"*. The work presented in this manuscript describes the view we acquired through numerous discussions and collaborations we had with researchers and practitioners around the world. Amongst them, we acknowledge here our closest academic collaborators, the Trame team at ESEO Angers, our colleagues in the SHARP department (Lab-STICC), as well as the postdoctoral fellows, the PhD candidates, the research engineers, and the students we have supervised. Furthermore, we appreciate the high-quality exchanges that we could have with practitioners from private companies, such as PragmaDEV, CS/SI, Clearsy, SNCF, and Naval Group.

Table 1.1 lists the PhD candidates with whom we have collaborated. The table shows the subject, the supervision team, the funding, the supervision rate, and our co-authored publications. We particularly want to acknowledge the long-term contributions of Luka Le Roux, a collaborator since 2013, first as a research engineer, then as a PhD candidate, and now as a postdoctoral fellow. The conceptual design of the G∀min∃ SLI, presented in this manuscript, became reality through the significant contributions of Valentin Besnard, who received the Accessit Prize of the CNRS GDR GPL for this PhD work[5]. Emilien Fournier challenged our understanding of model-checking at the algorithmic level and enabled us to dream of a scalable continuum between test and exhaustive verification through the fine-grained parallelism of reconfigurable architectures. Vincent Leilde, helped us to understand the multi-faceted importance of execution monitoring for reducing the cognitive burden during the diagnosis process. Our early collaboration with Jean-Philippe Schneider helped us understand the importance of dynamic interfaces for isolating the data from the analysis algorithms. Nicolas Tithnara Sun proposes influence engineering as a generalization of system engineering for sophisticated defense scenarios, which emphasizes the need for powerful reflexive operations in high-level specification languages. Lastly, Matthias Pasquier challenges one of our foundational hypotheses, the existence of a unique executable semantics, and we investigate the potential for semantic reuse for gradually introducing abstraction-based execution monitoring.

Besides PhD candidates, we had the pleasure to collaborate with several postdoctoral fellows and research engineers. Figure 1.4 summarizes these collaborations through a temporal vue (gantt-like). The blue bars show postdoctoral fellowships, the green bars represent research engineer contracts. Luka Le Roux and Vincent Leilde also appear in this figure, their PhD contract is shown in yellow. Our collaboration with Sebastien Tleye, in continuation of our PhD work on the model-driven physical design for reconfigurable architectures, emphasized the importance of designing model-agnostic analysis algorithms[58]. Our contributions on language-agnostic execution monitoring, discussed in this manuscript, can be seen as an instance of this more general setup. Jean-Charles Roger contributed to the software architecture of the OBP2 research prototype. During the postdoctoral fellowship of Fahad Golra we started investigating heterogeneous

[4]Seeing problem solving as a tree is nothing but a coarse approximation, in reality some nodes are shared and backlinks can arise, thus getting to a graph representation. Moreover problem decomposition is a highly subjective matter, naturally leading to multiple approaches for solving identical problems.

[5]https://gdr-gpl.cnrs.fr/node/444

Table 1.1: PhD candidate co-supervision from 2013 to 2022.

	Funding	Supervision Rate	Publications
Cyber-Security Certification and Embedded Operating System for IoT *Considération explicite d'un système d'exploitation embarqué dans un processus de certification de cybersécurité* **M. Pasquier** *co-supervision with L. Lagadec, M. Brun and F. Jouault*	CIFRE ERTOSGENER	30%	[92, 82]
Hardware Acceleration of Safety and Liveness Verification on Reconfigurable Architectures [93] *Accélération matérielle de la vérification de sûreté et vivacité sur des architectures reconfigurables* **E. Fournier** - soutenue 07/2022 *co-supervision with L. Lagadec*	Région Bretagne	75%	[24, 22, 23]
Systems Modeling and Formal Analysis for Advanced Persistent Threats [94] *Modélisation et Analyse Formelle de Modèles Système pour les Menaces Persistantes Avancées* **T. N. Sun** - soutenue 05/2022 *co-supervision with R. Mazo, P. Dhaussy and J. Champeau*	Pôle d'Excellence Cyber	75%	[13, 12, 56]
EMI: An Approach to Unify Analysis and Embedded Execution with a Controllable Model Interpreter [95] *EMI : Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable* **V. Besnard** - soutenue 12/2020 **Accessit Prix de Thèse du GDR GPL** *co-supervision with P. Dhaussy and M. Brun*	CIFRE Davidson Consulting	35%	[1, 17, 16, 25, 15] [26, 27, 14, 28, 57]
A Diagnosis Support for Formal Verification of Systems [96] *Aide au diagnostic de vérification formelle de systèmes* **V. Leilde** - soutenue 11/2019 *co-supervision with P. Dhaussy and V. Ribaud*	Région Bretagne		[29, 30, 31]
Critical embedded system verification, a non-intrusive approach to divide the initial challenge into a sound set of smaller ones [97] *Validation par parties et non-intrusive de systèmes embarqués* **L. Le Roux** - soutenue 11/2018 *co-supervision with A. Plantec*	PIA BGLE DEPARTS	80%	[4, 3, 21, 32, 33]
Towards an Efficient Approach for Model-checking with Cloud Computing [98] *Vers une demarche efficace de traitement du model checking dans les cloud computing* **L. Allal** - soutenue 04/2018 *co-supervision with G. Belalem and P. Dhaussy*	Univ. Oran, Algérie		[7, 6, 5, 34]
Roles : Dynamic Mediators Between System Models and Simulation Models [99] *Les rôles : médiateurs dynamiques entre modèles systèmes et modèles de simulation* **J.-P. Schneider** - soutenue 11/2015 *co-supervision with L. Lagadec, Eric Senn and J. Champeau*	DGA		[35, 36]

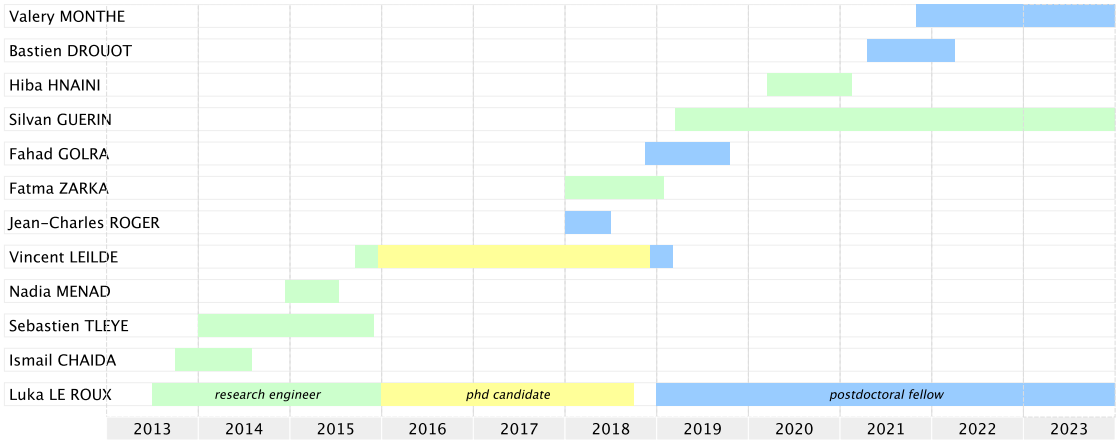


Figure 1.4: Collaborations with postdocs and research engineers at ENSTA Bretagne

language composition with the G∀min∃ SLI [18]. Hiba Hnaini and Bastien Drouot contributed to the experimental investigation of using heterogeneous verification for cyber-security in the context of autonomous vehicles [37]. Finally, with Silvain Guerin, we are currently investigating heterogeneous refinement mappings for establishing a link between legacy executable code and formal specifications.

Finally, I want to acknowledge the contributions of our undergraduate and master students. They have contributed significantly to the maturation of both our ideas and research tools. They have been our alpha-testers for finding bugs that we could not imagine. They have been our teachers, showing us the limits of our understanding. They have been our scouts, helping us identify some interesting research problems. The internships of Khaoula Es-Salhi and Rim S. Boudaoud contributed to the experiences [38] that led to the PhD of Vincent Leilde. Nicolas Gunepin and Michael Rigaud, have initiated the work which led to the PhDs of Luka Le Roux and Valentin Besnard, respectively. Lastly, the work of Riwan Cuinat on projectional editing TLA+ specifications [39] showed that by investing some effort we might be able to lower further the cognitive effort of writing and manipulating formal specifications.

1.6 Grants and Projects

The work presented in this manuscript was made possible with the financial support offered by international, national, and regional research grants, as well as bilateral contract with industrial partners. Figure 1.5 provides an overview of the research grants along with the funding sources. The main funding sources were:

- European Regional Development Fund (ERDF),
- Banque Publique d'Investissement (BPIfrance),
- French National Research Agency (ANR),
- French Directorate General of Armaments (DGA), through the Defence Innovation Agency (AID) since 2018,
- French Directorate of Civil Aviation (DGAC)

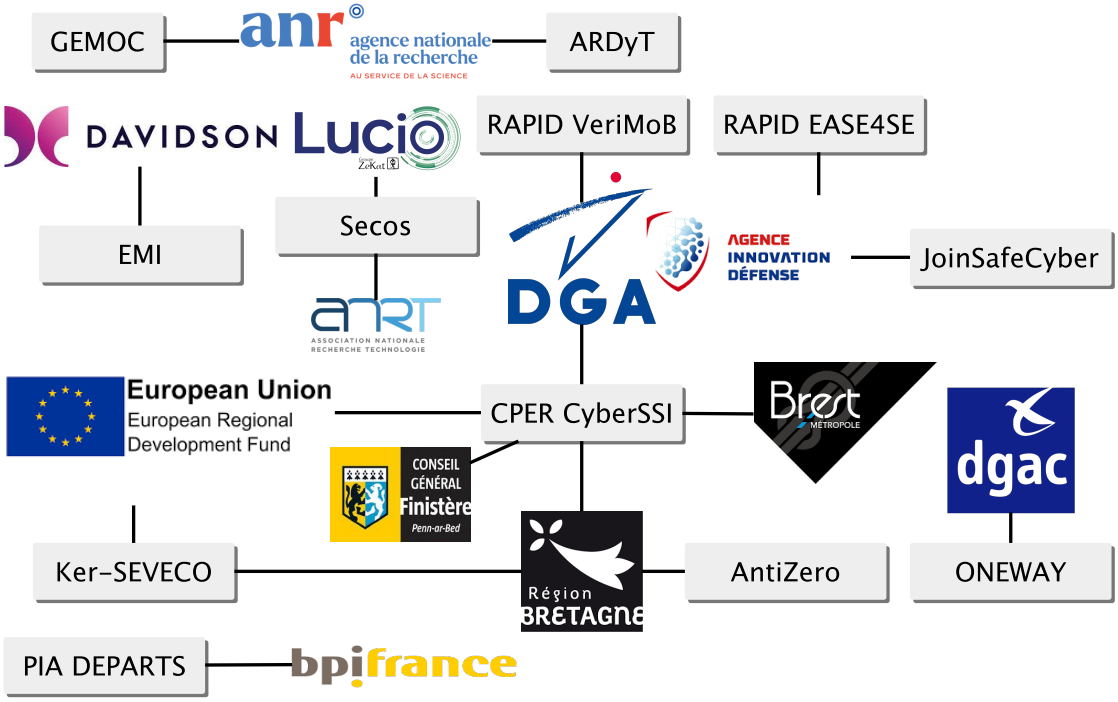


Figure 1.5: Research grants and funding sources

- French National Association of Technological Research (ANRT)
- Brittany Region
- Departmental Council of Finitère
- Brest Métropole,
- Davidson Consulting
- Lucio, groupe ZeKat

Besides the financial support, these grants provided the perfect context to challenge our understanding, uncover new problems, experiment, polish our ideas and transfer our solutions to practitioners in the industry. The PIA project DEPARTS (2013-2018) planted the seed that gave rise to the work presented here. The problem addressed was the heterogeneity and reuse of (ideally formal) specifications for reliable system design. In the consortium our research focus was the management of heterogeneity for model-checking, the solution prescribed was *straightforward*: "convert everything[6] to the common intermediate language Fiacre[88] (introduced in the FUI project TOPCASED), for which we have model-checkers".

Our concomitant implication in two other ANR projects GEMOC[7] (2012-2016) and ARDyT[8] (2011-2015) gave us a broader understanding of the problem. The ANR GEMOC project introduced an execution semantics axis, with the added complexity of designing a language-agnostic

[6]Some of the specification languages retained by the DEPARTS consortium where UML, Scade, B, AEFD[86]
[7]https://gemoc.org/ins/
[8]http://ardyt.irisa.fr/

coordination layer along with early validation and verification tools. At this stage, the problem becomes a lot more complex, due to the horizontal heterogeneity (the model can be expressed in multiple languages). Moreover, the coordination language is rather complex so the mapping to Fiacre[88] is not trivial. Furthermore, defining an operational semantics (GEMOC) for debugging and execution and then a denotational semantics (DEPARTS) for model-checking, seemed to ask a lot from the language engineers. Especially since the denotational semantics obtained is most often just a transformation of the operational semantics in another language (at the same abstraction level). Why not embrace this horizontal heterogeneity, and create a semantic bridge toward the analysis tools? Focused on the design of fault-tolerant and self-adaptable execution platforms, the ANR ARDyT project, brought into focus the problems related to hardware verification and the inherent vertical heterogeneity introduced by the mapping application-architecture. The plot thickens, from the formal verification perspective not only do we ask for denotational semantics (through the mathematical objects of the Fiacre language) for each language used by the application specification, but we also need that these semantics account for the architectural variability, since the applications might be mapped on multiple architectures, each bringing a different set of hypotheses.

The CPER CyberSSI (2015-2021) supported both the acquisition of a high-performance experimental platform and offered the context for applying our results to cyber-security problems. The cyber-security context adds yet a new axis to the verification problem, the robustness of the system to *intentional environment aggressions*.

The support of Davidson Consulting, EMI project (2017-2020), enabled us to explore the unification of model analysis (debugging, model-checking, runtime monitoring) and embedded execution, in the context of a rather challenging executable DSL, the Unified Modeling Language (UML).

The French Directorate General of Armaments (DGA) supported both the EASE4SE (2017-2019) and VeriMoB project (2018-2020) which offered us the opportunity to apply the G$\forall$min$\exists$ approach for the analysis of business processes in the NATO architectural framework. In this context, the collaboration with PragmaDEV (VeriMoB) led to the design of PROCESS, a commercial BPMN analysis tool[9]. The EASE4SE project, in collaboration with Sodius, explored the semantic-level composition of business process models with existing discrete-event simulation environments. These research efforts continue in the project ONEWAY (2021-2023) with the support of the French Directorate of Civil Aviation (DGAC), which addresses the digitalization of the product development strategies of complete aircraft systems. Three research directions are pursued: *1)* the formalization of a diagnosis language bound to the modeling semantics; *2)* the compositional integration of temporal constraints in business process models; *3)* the exploration of the continuum test – exhaustive verification, at the algorithmic level.

The AnaMenace (2017-2020) and the JoinSafeCyber projects (2019-2021), both supported by the Defense Innovation Agency, focused on the design of a tool-supported methodology for reliable system design in the context of advanced persistent threats. In this context, we have focused on two orthogonal problems: *1)* addressing the limitations of current system engineering methodologies for the analysis of complex cyber defense scenarios; *2)* the diagnosis of executable models through a compositional toolflow. The collaboration with ESEO Angers led to the design of the AnimUML environment[10] that offers an interactive toolkit for playing with the dynamics of partial UML specifications while they move towards embedded executables.

The Ker-SEVECO project, supported by the Brittany Region and the European Regional Development Fund, further explored a globally-asynchronous locally-synchronous (GALS) approach to heterogeneous verification for the formalization and security analysis of legacy code.

[9] http://www.pragmadev.com/product/process.html
[10] http://animuml.obpcdl.org/AnimUML.html

This approach can be seen as a restricted form of refinement mapping, where the legacy code (considered locally-synchronous) is wrapped in a specification (globally-asynchronous), which paves the way towards property verification by model-checking.

Amongst the monitoring tools that we consider, model-checking is the most promising but the most challenging, due to its intrinsic state-space explosion problem. The DEPARTS, CPER CyberSSI and the AntiZero (2017-2020) grants allowed us to explore this challenge at an algorithmic level. These efforts led to the discovery of novel algorithms that exploit the fine-grained observability of the semantics as well as the inherent parallelism of reconfigurable architectures. More recently the collaboration with Lucio (2020-2023), supported by the ANRT, allows us to further investigate the composition of abstract semantics with existing executable semantics for improving the scalability of model-checking in the context of vertical heterogeneity.

1.7 Conclusion

This chapter overviews almost ten years of research efforts focused on exploring the boundary between executable specification languages and behavioral analysis tools. The specification designer needs domain-specific languages to capture the set of behaviors required by the client's desired capability. This led to the creation of a large number of executable specification languages. A wide variety of tools (language monitors) are needed, initially, to improve the confidence that the specification correctly captures the desired functionality, and later, to prove that a given system correctly implements the specification. However, building a new set of language-specific tools for each new domain-specific language is not sustainable. The following question sets the scene: "How to bridge the gap between these domain-specific formalisms, geared towards domain experts, and the formal verification tools, geared towards mathematicians?" After discussing some of the scientific challenges in section 1.2, section 1.3 sketched a new semantic-level answer, dubbed G∀min∃, which promises a modular and compositional approach for building language monitors. Our contributions are presented along three axes: *1)* the *formalization* of the approach, which strives for modularity and compositionality; *2)* the *evaluation* through significant research artifacts, which led to contributions in the model-driven engineering community; and *3)* the *new opportunities*, which led to the invention of new verification techniques.

An overview of our research strategy, rooted in the design science approach, was presented, in section 1.4, which emphasized both the importance of a prototype-driven approach in our context and the complexity induced by problem decompositions in the larger context of scientific research.

In section 1.5, we have acknowledged the most fruitful collaborations with researchers, practitioners, and students around the world that pushed us farther, helping us shape our vision. We could not realize our research vision without the financial support of numerous public and private grants that were discussed in section 1.6.

The following chapter, chapter 2, dives deeper into G∀min∃ Semantic Language Interface, our underlying contribution, that structured the last decade of our research career, and which, continues to inspire us, opening the doors for more ambitious perspectives, briefly discussed in chapter 3.

Chapter 2

The G∀min∃ Semantic Language Interface and Applications

Contents

The contents of this chapter is adapted from the following articles:

- Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "Unified verification and monitoring of executable UML specifications". In: *Software and Systems Modeling* 20.6 (Dec. 2021), pp. 1825–1855. ISSN: 1619-1374.

DOI: 10.1007/s10270-021-00923-9. URL: https://doi.org/10.1007/s10270-021-00923-9

- Matthias Pasquier, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. "Practical Multiverse Debugging through User-defined Reductions. Application to UML Models." In: *Proceedings of the 24rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '22. 2022
- Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "Modular Scheduling for Verification & Embedded Execution." In: *submitted to Software Testing, Verification and Reliability* (2022)

This chapter introduces the backbone of our contributions, namely the G∀min∃ Semantic Language Interface (SLI). This observation and control interface isolates the language semantics from the analysis tools and enables the modular specification of several generic semantic operators, which eases the creation of specialized tools. Conceptually the G∀min∃ SLI exposes the underlying semantics through a transition system metaphor like the Plotkin structural operational semantics. From the tool's perspective, the SLI offers the formal vocabulary needed for interacting with the execution, while remaining independent of the implementation details of the subject language. The chapter is decomposed into four sections. Section 2.2 defines the G∀min∃ Semantic Language Interface. Section 2.3 introduces the most important generic operators, which streamline the link creation between the semantics and the specialized runtime environment needed either for analysis or embedded execution. Section 2.4, defines, based on the SLI, an unified debugging semantics that subsumes interactive, omniscient, and multiverse debugging. Section 2.5 further illustrates the strength of the approach by showing a novel way of considering scheduling in multiple execution setups ranging from embedded execution to model-checking. Section 2.5.6 concludes this chapter.

The definitions (listings) in this chapter are formalized using the lean theorem prover [100]. The lean source-code is available under a MIT license at https://github.com/teodorov/gamine.

2.1 Overview

This research effort is based on the observation that globally the software language community seems to be divided between two apparently incompatible endeavors: formalizing the language semantics for reasoning on the system dynamics and implementing performance-oriented runtime environments for language execution. In the first case, tools such as kframework[101] and gemoc studio[102] offer a rich set of tools for reasoning on the dynamics during execution. However, in these environments, the raw execution performance suffers. Furthermore, these approaches require reformulating the semantics using specialized domain-specific languages, which can incur a high development cost. On the other hand, the industry invests tremendous efforts in creating high-performance execution environments, which most often lack the support for reasoning. Furthermore, this situation can introduce semantic deviations between the real execution and the one seen by the analysis tools, which require language equivalence proofs to rule out. The model-checking community is probably the community the most impacted by this dichotomy. On one hand, the industry needs verification tools for real-world languages while on the other hand, the model-checking community pushes for highly constrained formal languages, which ease the verification problem.

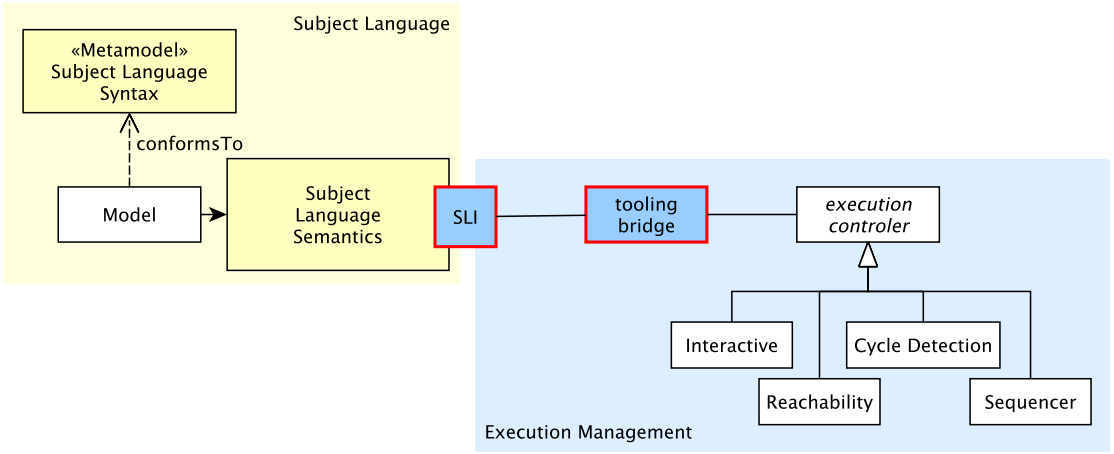


Figure 2.1: Global overview

The main research question that we ask then is: *Is it possible to compose existing execution environments with monitoring tools without reformulating the semantics?* For model-checking, the answer to this question seems to be positive as shown by research efforts such as Java PathFinder[103], LTSmin[104]. However, to enlarge the context to the more general scope of execution monitoring, some follow-up questions are:

- What are the required interfaces?
- What are the minimal set of services which should be offered by the executing environment to ease the monitoring process?
- How to formalize and implement high-performance tools independently from the language semantics?

This chapter addresses these questions by introducing the G∀min∃ semantic language interface (SLI). This interface extends the transition system semantics with a few specialized features that offer fine-grain observability and high level of control during execution. Based on this interface some common semantic-level operations are captured as composable operators, which enable the specification of many monitoring setups ranging from simple step-by-step execution to model-checking.

The conceptual decomposition of the problem, illustrated in 2.1, shows the SLI acting as a mediator between the subject language semantics (left, yellow box) and the execution management component (right, blue box) that effectively implements the language monitor. The subject language component illustrates the interplay between the main components of a language, the **semantics** captures the meaning of a **model** that conforms to a number of **syntactic rules** (a grammar or a metamodel). It is important to note that the SLI requires an open-style semantics, that exposes the execution steps descriptively, without invoking them. The execution management component isolates a tooling bridge from the execution controller. The tooling bridge component enriches the subject language semantics with the monitoring state and actions while preserving the open-style semantics. The execution controller explores the composite semantics offered by the tooling bridge to close the analysis loop. Here we consider four execution controllers: 1. an interactive controller, which allows a user to manually advance the execution; 2. a reachability controller, which performs a fixed-point computation using standard reachability algorithms,

such as BFS or DFS; 3. a cycle detection controller, which allows the detection of acceptance cycles in the underlying state-space; 4. a sequencer, which simply threads the execution, given a deterministic monitoring semantics.

2.2 The Semantic Language Interface

The G∀min∃ Semanting Language Interface (SLI) is defined around four main abstract types, which are mapped to the subject language internal data structures:

- `C`: the type of configurations. A configuration is a memory dump of all runtime data handled by an execution engine (i.e., its execution state) at a given time.
- `A`: the type of actions. An action is an abstract representation of a fireable transition or an execution step that enables going from one configuration to another.
- `E` : the type of expressions that can be evaluated by the subject language runtime.
- `L` : the type of expressions that need to be evaluated outside the scope of the subject language.

The configuration type `C` captures the variable-memory of the execution engine. This memory corresponds to the valuation of the dynamic variables. A configuration can be extended to capture the whole memory image of a program, including the constant-memory and the code-memory. However, for simplicity, in our experiences the dynamic memory is sufficient for capturing the dynamic program evolutions through a series of configurations. In our context, the constant and code memories do not change during execution, and they can always be retrieved from the runtime, or they can be computed by loading the program. Our approach does not constrain the internal representation of the configuration, we simply handle it as an opaque type, which can be inspected only by the evaluation services offered by the E language.

The action type `A` represents the executable steps of the subject language. Typically, they can be captured as a continuation, which, for improving observability, can be enriched by metainformation (such as line-numbers, links towards the AST nodes, etc.). In practice, though, dynamically building continuations can be costly. To accommodate this constraint, we again rely on an unconstrained opaque abstract type, which can be mapped to the subject language implementation. Similarly, to the configurations, the action can only be inspected by the evaluation services of the E language.

The `E` abstract type encodes expression in a diagnosis-specific language exposed by the implementation of the subject language. In the simplest cases, this type is interpreted directly by the expression evaluator of the subject language. However, for improving the observability of the execution of the subject language expression terms could be extended. These extensions can allow, for instance, to introspect the internal state of the execution runtime, even if the subject language does not include syntactic constructs for such operations. A typical example in this category will be inspecting the evaluation stack or the internal state of communication buffers.

The `L` abstract type captures the set of expressions that need external "services" for evaluation. For typical specification languages, this set is empty. However, it is needed for languages that depend on external inputs for progressing (i.e. a property language). This type is rather particular to property languages, which have their semantics dependent on an underlying model language. The integration of this type in the SLI is motivated by our view that all subject languages can be specialized to play the role of a property language in the monitoring setup. Furthermore, in the case of high-level specifications, this type could ease the creation of heterogeneous refinement mappings[105], a subject of our future works.

```
structure SLI (C A E L : Type) :=
    (str        : STR C A)
    (evaluate   : E → C → A → C → bool)  -- Atomic proposition evaluator
    (collect    : C → A → set L)         -- Atomic propositions collector
    (accepting? : C → bool)
```

Listing 2.1: The semantic language interface definition

```
structure STR (C A : Type) :=
    (initial : set C)
    (actions : C → set A)
    (execute : C → A → set C)
```

Listing 2.2: The Semantic Transition Relation

Based on these types the semantic language interface is defined in Listing 2.1. The *STR* term, which is further detailed in Section 2.2.1, captures the underlying execution runtime viewed as an extended transition relation. The *evaluate* function enables the evaluation of diagnosis expressions on execution steps composed of the source configuration, the executed action and the target configuration. The *collect* function gathers the expressions needing external evaluation in the context of an action enabled in a source configuration. The *accepting*? function defines the set of accepting configurations in a verification context.

2.2.1 Semantic Transition Relation

The G∀min∃ Semantic Transition Relation (STR) interface, presented in Listing 2.2, exposes the execution semantics of the subject language. This interface is the core of the SLI, offering a fine-grained view over the dynamic structure. As opposed to lower-level, model-checking specific interfaces [106, 104], the STR interface reifies the execution steps. Furthermore, it enables multiple sources of non-determinism to accommodate a large panel of subject languages.

The `initial` function of the STR captures all possible initial configurations (`set C`) of the model execution. Support for initial state non-determinism is specially required by high-level specification languages, such as TLA+ [107], which express the initial states as a predicate. Figure 2.2 illustrates this point through a simple automaton with multiple initial states ($s0$, $s1$), denoted as usual by the arrow emanating from a black point.

The `actions` function returns all available actions (`set A`) enabled from a given configura-

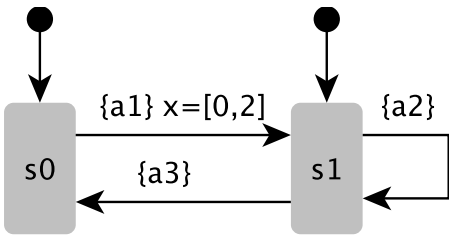


Figure 2.2: Simple automaton with initial, action and execution non-determinism.

tion (`C`). Seen from the perspective of SOS semantics[87], the action function returns the set of transition rules for which the current configuration satisfies the premises. This exposes the rule non-determinism of the executed model. In concurrent languages, the non-determinism arises at this level from the parallel composition of execution units (having multiple execution units ready to execute). Furthermore, some specification languages relax the determinism constraint even at the execution unit level, allowing multiple actions originating from the same execution unit. The typical example for this second case is an automata-based language, which can allow non-exclusive guards on the transitions emanating from the same source state. In Figure 2.2 the $a2$ and $a3$ actions are both enabled in the state $s1$. During deployment concurrent actions from multiple execution units may be executed in parallel on different processing cores. Nevertheless, each action is seen as atomic from the perspective of the subject semantics, leading to consistent target configurations[1].

The `execute` function executes one action (`A`) in a source configuration (`C`) and retrieves a set of target configurations (`set C`). From the SOS perspective, the execute function simply unfolds the conclusion of the transition rule. This function allows for non-deterministic rule execution using a predicate as a return type. The non-determinism, at this level, is typical for specification languages that either relax operator semantics or provide specialized constructs which introduce execution non-determinism. Non-deterministic assignment is a typical example of a non-deterministic operator, illustrated by the action $a1$ in Figure 2.2, where $x = [0, 2]$ allows x to be assigned any value from the $[0, 2]$ interval. Some of the most expressive synchronous communication semantics introduce execution non-determinism by either allowing unidirectional handshakes [108] or n-to-m directional synchronizations [88]. The usage of shared variables in these cases can lead to execution non-determinism due to the indeterminate ordering of the synchronous actions. Note that most implementation languages strictly prohibit execution non-determinism either syntactically or through implementation choices, which most often resolve the non-determinism by arbitrarily chosen policies. The most common example is the undefined behaviors of the C language semantics, which are somewhat resolved by most compilers [109].

2.3 Generic Monitoring Operators

In this section, we define 6 generic monitoring operators, which offer a reusable base for multiple verification and execution setups. The definition of these operators is based on the G∀min∃ SLI that isolates them from the potentially intricate details of the subject language semantics. The section starts by defining a semantic filtering operator (Section 2.3.1), which enables semantic-aware scheduling (Section 2.3.2) for partially resolving the ”actions” non-determinism. The asynchronous composition and synchronous product (Section 2.3.4) binary operators are discussed in Section 2.3.3, and Section 2.3.4, respectively. Finally, the mapping between the STR to a model-checking specific transition relation is defined in Section 2.3.5.

2.3.1 Filter

Given the SLI formalization, we can define a filter operator that applies a partial filtering over the set of actions issued by the STR. The selection of actions to filter is made according to a filtering policy that defines which actions are forwarded and which action are removed from the set. The filtering policy, defined in Listing 2.3, has an execution state, which is denoted by the S type in our formal description.

[1]The language semantics may define synchronization points between the execution of multiple steps. In this case, the synchronous execution steps need to be considered as a single action because only the global target configuration is consistent.

```
structure FilteringPolicy (C A S : Type) :=
    (initial : S)
    (selector : A → Prop)
    (apply : S → C → set A → set (S × A))
    (subset² : ∀ s c 𝔸 (sa ∈ (apply s c 𝔸)),
        prod.snd sa ∈ 𝔸)

def StatelessFilteringPolicy (C A : Type) := FilteringPolicy C A unit
```

Listing 2.3: Filtering policy definition

```
def filter (C A S: Type) (m: STR C A) (s: FilteringPolicy C A S) : STR (C × S) (S × A)
:= {
    initial := {cs | ∀ (c ∈ m.initial), cs = (c, s.initial)},
    actions := λ cs,
        let toFilter := { a ∈ m.actions cs.1 | s.selector a },
            toForward := { fa | ∀ a ∈ m.actions cs.1,
               ¬ s.selector a → fa = (cs.2, a)}
        in (s.apply cs.2 cs.1 toFilter) ∪ toForward,
    execute := λ cs sa,
        let
            r := m.execute cs.1 sa.2
        in {x | ∀ c ∈ r, x = (c, sa.1)}
}
```

Listing 2.4: Filter operator definition

The `initial` function returns the initial execution state (`S`) of the filtering policy. The `selector` function enables to select actions on which the filter is applied. In some cases, the filter has only to be applied on some actions and not on others (e.g., it may happen that we only want to apply the filter on actions coming from the system and not on those coming from the environment). Given the selected actions (`set A`), a configuration (`C`), and the execution state of the filter (`S`), the `apply` function defines which actions must be forwarded while others are removed. For each forwarded action, this function also returns the new execution state of the filtering policy. As shown by the `subset` constraint, the returned set of actions (`sa`) is expected to be a subset of the action set ($\mathbb{A}$) given as input. It is also important to notice that to be used in model-checking, the filtering policy should be deterministic, which explains why it has only one initial execution state. From a given execution state and a set of actions given as input, the scheduling policy must always return the same set of actions as output. Moreover, using the definition of the `FilteringPolicy`, it is also possible to define a stateless version, named `StatelessFilteringPolicy`, without an execution state.

Based on the definition of the filtering policy, Listing 2.4 provides the formal definition of the filtering operator. A filter takes as inputs an STR and a filtering policy and returns a new STR for which actions have been filtered while keeping track of the filtering policy state.

For each initial configuration obtained through the STR interface, the `initial` function returns a pair containing the initial configuration of the STR and the initial execution state of the filtering policy. The `actions` function begins by separating actions on which the filtering policy has to be applied (`toFilter`) and actions that have only to be forwarded (`toForward`). Then, the filtering policy is applied on the `toFilter` set and the resulting actions are returned

```
structure SchedulingPolicy (C A S : Type) extends (FilteringPolicy C A S) :=
    (unique : ∀ s c 𝔸 (a ∈ (apply s c 𝔸)) (b ∈ (apply s c 𝔸)), a = b)

def StatelessSchedulingPolicy (C A : Type) := SchedulingPolicy C A unit
```

Listing 2.5: Scheduling policy definition

```
def scheduler (m : STR C A) (s : SchedulingPolicy C A S) : STR (C × S) (S × A) :=
    filter C A S m ↑s
```

Listing 2.6: Scheduler operator definition

along with the elements of the `toForward` set. The `execute` function delegates the execution of a given action to the subject STR (m) and computes the target configurations which are composed of the target configurations obtained through the STR and the resulting execution state of the filtering policy.

2.3.2 Scheduler

Among a set of actions, the scheduler selects one of them as the next execution step for model execution. The choice of this action depends on a scheduling policy, presented in Listing 2.5, which is a specialization of the filtering policy.

The scheduling policy adds the `unique` constraint to the filtering policy to ensure that the `apply` function returns a unique action rather than a set. Similarly to the `StatelessFilteringPolicy`, it is possible to define a stateless version of the scheduling policy, the last line in Listing 2.5.

Listing 2.6 provides the formal definition of our scheduler operator. The scheduler takes as input a STR and a scheduling policy and returns a new STR that has only one available action at a time, the action selected by the scheduling policy. This definition shows how the scheduler can be easily defined as a filter. Note the upcast (↑s) of the scheduling policy to a filtering policy.

In our approach, we see the scheduling problem as a parametric filter on the possible execution steps available for execution at a point during the application's lifetime. As we focus on single processor targets, we expect that the scheduler selects at most one execution step. Please note that the language semantics can provide an empty set of possible execution steps, due to deadlocks or termination, in which case the scheduler trivially returns no further execution steps. Also, note that the scheduler only solves the non-determinism induced by the concurrency between the execution units. Further work is thus necessary to ensure that each execution unit is deterministic when considering system deployment. Nevertheless, this determinization process is not necessary for preliminary analysis phases.

2.3.3 Interleaved Composition

The interleaved composition is a binary composition operator that composes two processes without synchronization. In our context, this asynchronous composition operator effectively merges the actions of two STR.

The asynchronous composition operator takes as input two STR and returns the asynchronous composition of both inputs as another STR, as defined in Listing 2.7.

```
def interleave { C₁ A₁ C₂ A₂ : Type }
    (lhs : STR C₁ A₁)
    (rhs : STR C₂ A₂)
:   STR (C₁ × C₂) (A₁ ⊕ A₂) :=
{
initial  := {(c₁, c₂) | ∀ (c₁ ∈ lhs.initial) (c₂ ∈ rhs.initial)},
actions  := λ c₁, c₂), { a | ∀ (a₁ ∈ lhs.actions c₁) (a₂ ∈ rhs.actions c₂),
                        a = sum.inl a₁ ∨ a = sum.inr a₂
                },
execute  := λ (c₁, c₂) a, { c' |
                    match a with
                        | (sum.inl a₁) := ∀ c₁' ∈ lhs.execute c₁ a₁, c' = (c₁', c₂ )
                        | (sum.inr a₂) := ∀ c₂' ∈ rhs.execute c₂ a₂, c' = (c₁ , c₂')
                    end
            }
}
```

Listing 2.7: The definition of the interleaved composition operator.

The `initial` function returns the cartesian product of initial configurations of both STR. The `actions` function joins the actions of both STR in a sum type ($A_1 \oplus A_2$), representing the disjoint union of the underlying action sets (A_1 and A_2). To execute an action, the `execute` function checks from which STR the action is coming and executes it on the corresponding STR. Each target configuration returned is a pair containing one target configuration of the STR on which the action has been executed and the current configuration of the other STR.

2.3.4 Synchronous Product

The synchronous product composes two automata based on the intersection of their vocabulary. This is the backbone semantic operator for computing language intersection queries.

Considering two automata

$$\mathcal{A}_1 = (S_1, \Sigma_1, \rightarrow_1, I_1, F_1) \quad \text{and} \quad \mathcal{A}_2 = (S_2, \Sigma_2, \rightarrow_2, I_2, F_2)$$

where for each automaton A_i, S_i represents its set of states, Σ_i its vocabulary, $\rightarrow_i$ the transition relation, $I_i \subseteq S_i$ its initial states, and F_i its accepting states.

The synchronous product can be defined as follows

$$\mathcal{A}_1 \otimes \mathcal{A}_2 = (S_1 \times S_2, \Sigma_1 \cap \Sigma_2, \rightarrow_\otimes, I_1 \times I_2, F_1 \cap F_2)$$

where the transition relation of the product ($\rightarrow_\otimes$) is

$$\frac{s_1 \xrightarrow{e}_1 s_1' \quad s_2 \xrightarrow{e}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{e}_\otimes \langle s_1', s_2' \rangle}$$

In simple automata formalisms (NFA or Büchi), this operator is rather straightforward. It simply exposes composite execution steps obtained by constraining the two operands to advance together on their common vocabulary. However, the application of this operator, for model-checking, on arbitrary transition relations poses a number of challenges. One of these challenges is the definition of the "common vocabulary" between the model and the property being verified. Furthermore, while the synchronous product operator is symmetric, there is an asymmetry

```
def sychronous_product (C₁ C₂ A₁ A₂ L : Type)
   (lhs : STR C₁ A₁)
   (ape : L → C₁ → A₁ → C₁ → bool)
   (rhs : STR C₂ A₂)
   (apc : C₂ → A₂ → L)
: @STR (C₁ × C₂) (A₁ × A₂) := {
initial := { (c₁, c₂) | ∀ c₁ ∈ lhs.initial c₂ ∈ rhs.initial },
actions := λ (c₁, c₂),
        { (a₁, a₂) | ∀ a₁ ∈ lhs.actions c₁
                       a₂ ∈ rhs.actions c₂
                       t₁ ∈ lhs.execute c₁ a₁,
                   evaluate (collect c₂ a₂) c₁ a₁ t₁
        },
execute := λ (c₁, c₂) (a₁, a₂),
        { (t₁, t₂) | ∀ t₁ ∈ lhs.execute c₁ a₁
                       t₂ ∈ rhs.execute c₂ a₂
        }
}
```

Listing 2.8: Asymmetric synchronous product between a model and a property.

between the model and the property. Namely, the property can be seen as an abstraction of the model that should somehow be connected to the model (to follow its evolution). Thus, conceptually, the property should speak the language of the model. In the literature, this problem is typically solved through Kripke structures[110]. In this approach, the property specification specifically integrates queries referring to the underlying model, known as *Atomic Propositions*. The underlying transition system (induced by the model) is interpreted from the perspective of these atomic propositions to produce a Kripke structure. The Kripke structure can then be viewed as an automaton, with the vocabulary Σ defined by the set of valuations of the atomic propositions.

In [111], the authors extend this framework for model-checking to allow the queries both on the model states and events (actions). This extension allows expressing the properties in a more concise manner and enables a more straightforward verification procedure, which alleviates the need for converting between state-based and event-based encodings of the model-checking problem. In TLA+ [107] the same framework is extended to allow "queries" on pairs of states, which enables to logically relate states. Since the STR reifies both the configurations and the actions of the subject language semantics, these extensions are natural in our context. The *evaluate* function accommodates this setup by allowing queries on complete execution steps (two related configurations and the action that relates them).

In our case, the synchronous operator takes as inputs two STR-based execution engines: one for the system model and another for the property model. For this definition, we assume that the model execution is the `lhs` term and the property is the `rhs` term. One evaluation function is also required for each execution engine, connected by the labels represented by `L`. These functions enable the property semantics to retrieve the valuation of the atomic propositions from the model execution semantics.

The `synchronous_product` of the system model STR (`lhs`) with the property model STR (`rhs`) defines a new STR, defined in the Listing 2.8.

The initial configuration of the composition (`initial`) is the product of the initial configurations of the lhs and the rhs.

```
structure TR (C : Type) :=
    (initial    :       set C)
    (next       : C ->  set C)
    (accepting? :       set C)
```

Listing 2.9: Transition relation with accepting states

```
def STR2TR (C A: Type) (str : STR C A) (sa : C -> bool) : @TR C  :=
{
    initial     := str.initial,
    next        := λ c, { t | ∀ a  ∈  str.actions c, t  ∈  str.execute c a },
    accepting?  := sa.accepting?
}
```

Listing 2.10: Converting from STR to a transition relation

To build synchronous actions (`actions`), the property semantics executes its `collect` function on all its available actions and returns the list of atomic propositions needing evaluation. These predicates are evaluated with the `evaluate` function on each execution step of the model (the tuple source configuration c_1, action a_1, target configuration t_1). The step a_1 has to be executed to get the target configuration of the system. Using the valuation of the atomic propositions, the guards of the property can now be fully evaluated. If a guard evaluates to true, it means that the action of the property (a_2) can be synchronized with the one of the model (a_1). As a result, we get a synchronous action corresponding to the tuple (a_1, a_2).

To execute a synchronous action (`execute`) we simply dispatch each primitive action to the corresponding STR. Note that, the order of execution is irrelevant since the model and the property do not share state.

2.3.5 Conversion to a Transition Relation

The introduction of the STR, as the main component of our semantic language interface, enables the definition of rich semantic operators, which can finely observe the semantics of the subject language. Sometimes, however, it is better to hide the complexity of the underlying execution engine. One of these cases is binding to model-checking algorithms, which require a simpler transition relation, as presented in Listing 2.9. This transition relation exposes only how the configurations are connected irrespective of the execution details. Besides the relation between states, the only ingredients needed for model checking is knowing the accepting states, and their interpretation (different between NFA and ω automata). In the following, we focus only on the first aspect, since the semantic interpretation can be provided by the instantiation context.

The conversion from a STR to a transition relation is defined in Listing 2.10. The *initial* states are the same, the relation between the states (*next*) is computed by collecting the target state obtained by executing the actions enabled in the source state. Finally, to identify the final state, the *accepting*? function of the SLI is mapped on the *accepting* predicate of the transition relation.

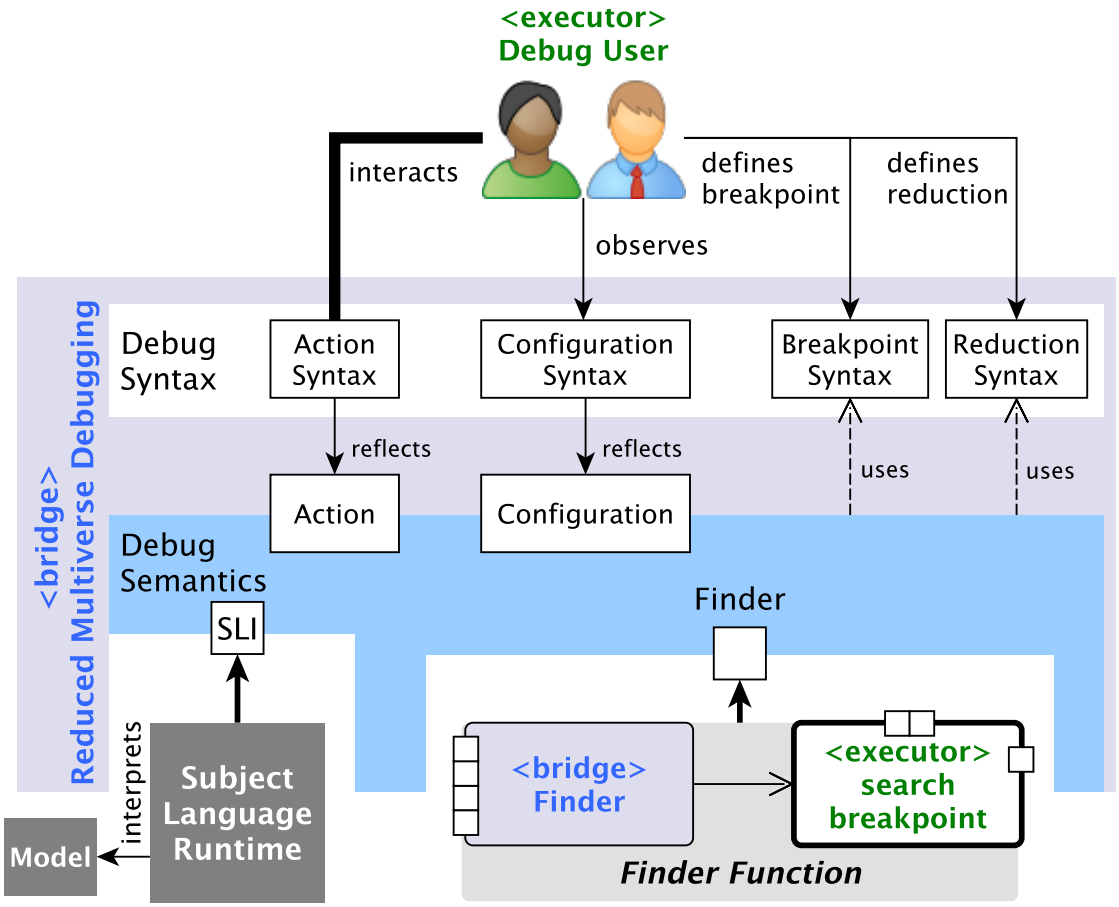


Figure 2.3: Architectural overview of the G∀min∃ unified debugger

2.4 Monitor Specification : The G∀min∃ Unified Debugger

Debugging is an important part of the development life cycle. By allowing the developer to explore programs interactively, it can be useful to find faults as well as simply get a better comprehension of their inner workings. As the complexity of programs increased, this process has also evolved to adapt to these new difficulties.

Omniscient debugging [112] allows us to return back, on the execution trace, to previously visited configurations. Multiverse debugging [81] has been introduced to enable the exploration of concurrent actor-based formalisms. This technique becomes important for the debugging of multi-threaded actor systems, where the different execution schedules can hide bugs. For high-level specification languages, such as TLA+ and UML, this feature is necessary due to the intrinsic non-determinism, which besides scheduling allows capturing entire families of implementations. The K debugger, presented in [113], is similar to our approach. However, it requires the use of the kframework[101] for specifying the language semantics. Furthermore, it does not support multiverse debugging[81].

Figure 2.3 overviews the architecture of our debugger emphasizing: on one hand, the actions available to the end user; and, on the other hand, its modularity with respect to the subject

```
structure DebugConfig (C : Type) :=
    (current : option C)
    (history : set C)
    (options : set C)
```

Listing 2.11: The configuration of the G∀min∃ debugger

language semantics. As debugging remains a human activity, the user is represented as the "debug user" executor (controls the execution interactively). The interactions during debugging are classified as: the actions (the orders), observing the configuration (state) of the system, defining breakpoints, and defining reduction policies. The user actions include stepping through the system's behavior, jumping back to a previously observed configuration, selecting between a non-deterministic choice and running to a breakpoint from the current configuration. The breakpoint definition syntax allows the user to define the stop criterion. Lastly, the user can define a reduction policy that is used during breakpoint lookup.

The debugger semantics can be seen as a bridge between the user and the debugged model. In other words, the G∀min∃ debugger is defined as a monitoring wrapper over the subject language. The debugger is dependent on the subject language runtime which captures the dynamic semantics of the model. In our case, the subject semantics is encapsulated in a Semantic Language Interface (SLI), that provides language-agnostic observation and control services. Besides the subject language, the Figure 2.3 emphasizes the modularity of our approach and the importance of the breakpoint *Finder Function* by representing it as an independent unit. When debugging a deterministic model, this function unravels the single execution path possible. However, multiverse debugging forces this function to search through multiple non-deterministic branches, a typical functionality of model-checking, which brings the state-space explosion problem. The run-to-breakpoint action of the RMD is realized by calling the Finder Function passing the subject language runtime as a parameter along with the breakpoint and the reduction policy.

The debugger specification, presented in the following, abstracts away some implementation details, such as the structure of the history trace, the projection functions needed for the user interface, and the search strategy used for finding the breakpoints. In [82] the *Finder Function* uses directly the *evaluate* component of the subject language SLI for breakpoint evaluation during a reachability search. To better illustrate the flexibility of our approach, in this manuscript, we choose a more generic approach for defining the *Finder Function*. This is achieved by running the reachability search on the synchronous product between the subject-language runtime and a property-language runtime, which offers an interpretation of the breakpoints. The debugger itself exposes the SLI interface. Note that, for simplicity, in the following, we focus mainly on the definition of the STR component of the debugging SLI. Moreover, this section will not discuss the user-defined reductions presented in Figure 2.3. Nevertheless, the interested reader should refer to [82] for further details.

The specification is based on two specific types: the debugger configuration and the debugger actions. The debugger configuration, DebugConfig in Listing 2.11, is parameterized by the subject language configuration type (C) and simply captures the core state-variables of the debugger, namely the current state (*current*), the execution history (*history*), and the options for next configurations (*options*). The *current* configuration simply wraps the current configuration of the subject language in an option type. The *history* is the set of configurations encountered since the beginning of the debugging phase. The *options* variable is necessary because of the non-determinism of the execute function in the subject language. The *options* variable acts as a

```
inductive DebugAction {C A : Type}
| step : A → DebugAction
| select : C → DebugAction
| jump : C → DebugAction
| run_to_breakpoint : DebugAction
```

Listing 2.12: Debugging actions, the abstract syntax of the debugger

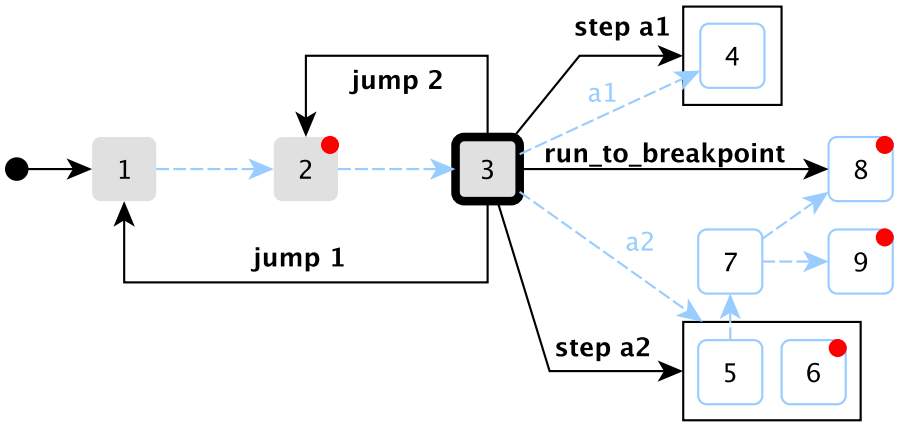


Figure 2.4: The G∀min∃ debug actions overlayed over a simple transition system (the dashed blue lines indicate the transitions)

temporary configuration buffer from which the next configuration can be selected.

Please note that we relax the type of the *current* variable to an option type. This is not necessary in practice. However, for the purposes of this presentation, it simplifies the specification by alleviating the need to ”arbitrarily” choose one configuration from the options to set as ”current” configuration. Furthermore, note that the *history* does not need more structure, and can simply be represented as a set, which subsumes multiple implementations as discussed later.

The abstract syntax of the debugger is captured by the DebugAction type, illustrated in Listing 2.12, which introduces 4 core debug actions. The DebugAction is parameterized over the configurations (C) and actions (A) of the subject language. The *step* action syntactically describes the intention of executing a particular action of the wrapped STR. The *select* action captures the selection step necessary for resolving the execution non-determinism of the wrapped STR. The *jump* action represents the intention of moving from the *current* configuration to an arbitrary configuration picked from the *history*. Finally, the *run_to_breakpoint* action represents the intention of running the underlying STR until a ”breakpoint” is found.

To gain some intuition on the G∀min∃ debugger, Figure 2.4 shows an example of the debug actions allowed by our specification. The figure overlays the debug actions over the configuration 3 (the rounded rectangle with a thick border) of a simple transition system (the dashed blue lines indicate the transitions). The black dot denotes the initial state. The red dots denote the configurations that satisfy the breakpoints predicate. The $a1$ & $a2$ labels on the blue transitions explicitly indicate the original actions which link the configurations. The execution of the $a2$ action is non-deterministic. Moreover, the configurations 3 & 7 have action non-determinism (two actions are enabled). The gray rounded rectangles show the configurations unrolled by the debugger up to the current configuration. From the current configuration (configuration 3) five

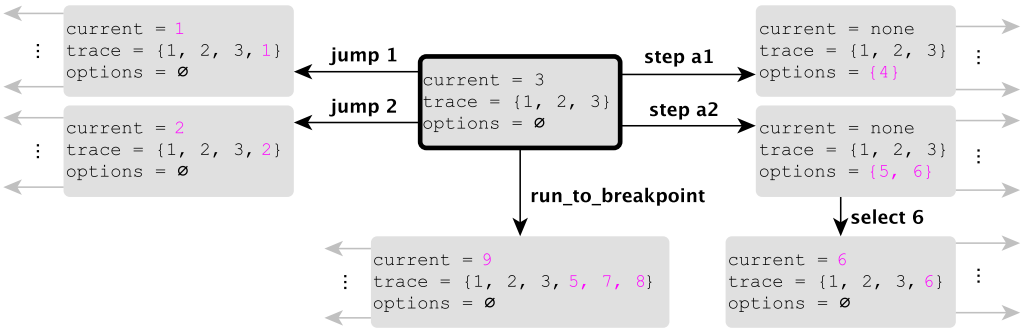


Figure 2.5: The debug transition system induced on the example in Figure 2.4

```
def Finder  (C A E : Type) := (Evaluate C A E) → STR C A → E → set C → list C
def Finder' (C A E : Type) :=                    STR C A → E → set C → list C
```

Listing 2.13: The signature of the counter-example finder function

actions are enabled: *a)* two *step* actions, which execute the $a1$ and $a2$ actions of the original transition system; *b)* two *jump* actions, which enable jumping back to one of the previously discovered configurations; *c)* the *run_to_breakpoint* action, which discovers the trace ($8 \leftarrow 7 \leftarrow 5 \leftarrow 3$) of the original transition system. Conceptually, any subset of all witness traces may be produced, as in [81]. For simplicity, our specification limits the scope of the run_to_breakpoint action to the first discovered witness. Thus, the configurations 6 and 9 are not seen by this action execution.

Figure 2.5 shows an excerpt of the transition system induced by our unified debugger specification, defined in Listing 2.14, on the original transition system in Figure 2.4. The gray arrows separated by $\cdots$ indicate the missing actions. Each rounded rectangle represents one instance of *DebugConfig*, with its three slots. The effects of the five debug actions are emphasized with the text fuchsia. The *step* actions set the options slot to the configurations obtained by executing the original action. The *jump* actions, change the *current* slot, extend the *history* and reset the *options* slot. The *run_to_breakpoint* action updates the *current* configuration and extends the *history* with the witness trace. The *select* action changes the current slot, extends the *history* and resets the *options* slot. In our example, it is used to choose configuration 6, between the two options (5 & 6) obtained by stepping with $a2$.

For simplicity, in the following, we consider a static set of breakpoints, which is obtained from the ambient environment. The breakpoints set is encoded using the E language, which corresponds to propositional logic over the diagnosis language (E) of the subject STR.

The strategy used for running the underlying STR is abstracted away by the *Finder* function; the signature is presented in Listing 2.13. Besides the type arguments, this function has three arguments: the *Evaluate* function of the underlying SLI, the STR that should be executed, the set of breakpoints, and the set of configurations from which to start the "execution". The return type is a witness, a list of configurations C showing a path from one of the breakpoints back to one of the start configurations. The *Finder'*, in Listing 2.13, results from the partial application of *Finder*. In section 2.4.2, we define this function modularly using the SLI.

```
def unified_debugger (C A E: Type) (o : STR C A) (evaluate : Evaluate C A E)
    (breakpoint : E) (finder : Finder C A E)
: STR (DebugConfig C) (@DebugAction C A) :=
{
   initial :=          debugInitial C A o,
   actions := λ dc,    debugActions C A o dc,
   execute := λ dc da, debugExecute C A L₀ o (finder evaluate) breakpoint dc da
}
```

Listing 2.14: The STR specification of the G∀min∃ debugger

```
def debugInitial (C A : Type) (o : STR C A) : set (DebugConfig C) :=
    {{ current := none, history := ∅, options := o.initial }}
  -- ⟨            none,          ∅,            o.initial ⟩ positional notation
```

Listing 2.15: The initial state predicate of the G∀min∃ debugger

2.4.1 A Semantic Transition Relation for Debugging

Based on the previous definitions the semantics of the debugger can be captured as a STR, illustrated in Listing 2.14. The *unified_debugger* STR takes 4 arguments: the debugged STR (o) along with the related APE function, the Finder function (*finder*) and the set of breakpoints. The return type is a new STR that wraps the original one while providing the debug actions.

Listing 2.15 shows the definition of the *debugInitial* function, which simply creates a singleton containing a *DebugConfig* with no *current* state, an empty *history* set, and the *options* variable set to the initial states obtained from the wrapped STR o.

The *debugActions* function, shown in Listing 2.16, generates the set of enabled actions in a *DebugConfig* dc. In our case there are four sources of actions:

1. the original action set oa, obtained from the subject STR o (only if the *current* slot is not empty). These actions are created through the *step* constructor.

2. the select action set sa, induced by the elements in the *options* slot, which are encapsulated by the *select* constructor.

3. the jump action set ja, induced by the *history* elements, which are encapsulated by the *jump* constructor.

4. the run_to_breakpoint action.

```
def debugActions (C A : Type) (o : STR C A) (dc : DebugConfig C) : set (DebugAction C A)
:= let
    oa := { step   a | ∀ c, dc.current = some c → ∀ a ∈ (o.actions c) },
    sa := { select c | ∀ c ∈ dc.options }
    ja := { jump   c | ∀ c ∈ dc.history },
in oa ∪ ja ∪ sa ∪ { run_to_breakpoint  }
```

Listing 2.16: The specification of the debugger ”actions” function

```
def debugExecute (C A E: Type) (o : STR C A)
    (finder : Finder' C A E) (breakpoint : E)
    (dc : DebugConfig C) (da : DebugAction C A) : set (DebugConfig C) :=
match da with
| step a            := { ⟨ none, dc.history, o.execute c a ⟩ |
                         ∀ c opt, dc.current = some c → o.execute c a ≠ ∅}
| select c          := { ⟨ c, dc.histoty ∪ { c }, ∅ ⟩ }
| jump c            := { ⟨ c, dc.history ∪ { c }, ∅ ⟩ }
| run_to_breakpoint :=
    match dc.current with
    | some c  := { ⟨ w, t, ∅ ⟩ |
        ∀ w l t, finder o breakpoint { c } = w::l → t = dc.history ∪ {x | x ∈ w::l}
    | none     := { ⟨ w, t, ∅ ⟩ |
        ∀ w l t, finder o breakpoint dc.options = w::l → t = dc.history ∪ {x | x ∈ w
    ::l}
    end
end
```

Listing 2.17: The specification of the "execute" function

The *debugExecute* function, presented in Listing 2.17, defines the execute step semantics of our *unified_debugger*. If the *current* slot is not empty (`dc.current = some c`) and the original action execution does not block (`o.execute c a` $\neq \emptyset$), the execution of the step creates a new configuration with the *options* slot assigned to the results of calling the original execute function. Otherwise, the execution of the step blocks. The *select* and *jump* executions are identical, the argument configuration is set in the *current* slot, the *history* is extended with the argument configuration and the *options* slot is emptied. The execution of a *run_to_breakpoint* is different based on the state of the *current* slot. If empty, then the *finder* function is invoked with the configurations from the *options* slot as starting point. If the *current* slot is not empty (`some c`), the finder starts the lookup for breakpoint from the configuration in this slot. If the finder locates a witness trace towards one of the breakpoints, a new configuration is created with the witnessing configuration (the configuration that satisfy the breakpoint condition) in the *current* slot and the *history* is extended with the witness (`dc.history` $\cup$ `{x | x ∈ w::l}`). Otherwise, the execution does not produce any configuration (it blocks).

Note that the execution function is deterministic, since it produces at most one new *DebugConfiguration* for each action. Furthermore, note that the blocking behavior of the *debugExecute* function is only a design choice. Nevertheless it prevents the introduction of "strange" debug configurations, like ⟨ `none, dc.history,` $\emptyset$ ⟩) which can be quit only with *jump* actions.

2.4.2 A Modular Finder Function

The *Finder* function, shown as a gray box in Figure 2.6, is the workhorse of any debugger. It enables the user to jump forward in time, over the transition relation, to specific "points" of interest (identified by breakpoint predicates). In deterministic sequential languages the *Finder* function corresponds to running the program until a breakpoint condition is satisfied. In general, however, the *Finder* needs to perform a reachability query on the transition system induced by the semantics. Relying on the functionalities exposed by the SLI, this functionality can be implemented in an ad-hoc manner by specializing any reachability algorithm to the peculiarities of the SLI interface. However, from our perspective, such an approach defeats the purpose of the SLI

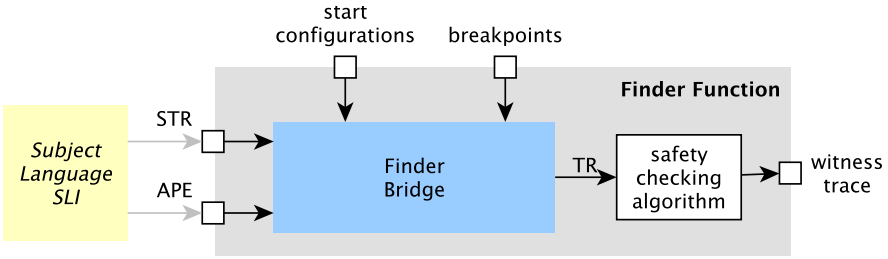


Figure 2.6: Overview of the *Finder* function

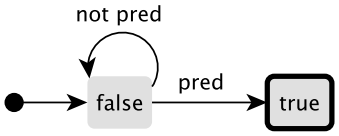


Figure 2.7: Breakpoint interpretation as a deterministic finite automata

interface, which aims at decoupling the semantics from the algorithm to enable their independent evolution. From this point of view, we should not compromise eighter on the semantics or on the reachability algorithm, but rather exploit the SLI facilities to build a bridge between the two, as illustrated in Figure 2.6 (the blue *Finder Bridge* rectangle). This section assumes a general *safety checking algorithm* (white rectangle in Figure 2.6) that operates on a transition relation and computes a counter-example. The rest of this section focuses on the modular specification of the *Finder Bridge* component, which can be achieved through our approach.

The first ingredient to consider is the subject language SLI (the yellow rectangle in Figure 2.6), from which we need the STR along with the atomic proposition evaluator (APE) functions. As seen in the previous sections, the STR captures the subject language semantics and exposes a local view based on the current configuration. The APE predicate extends the subject language evaluation facilities to enable queries over the execution steps. Querying full execution steps subsumes multiple usage scenarios by allowing a rich set of breakpoints, such as:

- Predicates related to the syntax tree (AST), the run-to-line is one example, where the line number is a proxy for the AST node.
- Predicates on the configuration (state predicates), the conditional breakpoint is an example, which is a conjunction of a state-predicate and an AST predicate.
- Predicates on the execution steps, that relate the source, the action, and the target configuration. An example of such a conditional breakpoint can be $x' = x + 1 \wedge \mathit{enabled}(\mathit{send})$, stating that the variable x in the source state is incremented by one and that the send AST-node is enabled by the action.

The breakpoints interpretation is defined through a deterministic finite automaton (DFA), illustrated in Figure 2.7, where *pred* represents a propositional logic expression using the Evaluate language for the atomic propositions (`E` in our formalization). Initially, the breakpoint is considered inactive (in the *false* state). During a system step, the breakpoint state either changes to *true* if *pred* holds on the step or remains inactive otherwise (*not pred*). The *true* state is the accepting state of the DFA. Moreover, the *true* configuration deadlocks, feature used to cut the execution path once the breakpoint is active.

```
def predicate_asserter (L₀ : Type) (pred : E) : STR bool (E × bool) :=
{
    initial := { false },
    actions := λ c, if c then ∅ else
        {(       pred, true),
        (  not pred, false)},
    execute := λ c a, { a.2 }
}
```

Listing 2.18: The *predicate_asserter* STR

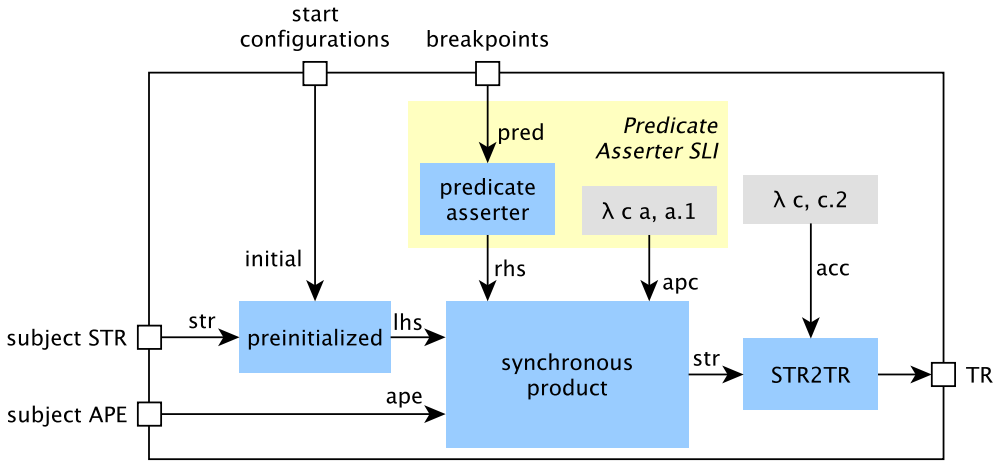


Figure 2.8: The architecture of the *Finder Bridge*

This deadlock interpretation is captured by the *predicate_asserter* STR, presented in Listing 2.18. The configuration type is bool. The action type is a pair (predicate, next-state). The initial set contains only the *false* state. The actions set is defined according to the current configuration c. If the DFA is in the *true* state, the action set is empty (`if c then ∅`), otherwise a set with the two possible transitions is returned. The execution of the action a in the configuration c is simply the projection of the $next - state$ part of the action pair (`a.2`). Besides the STR, the other SLI functions can be defined to obtain a complete SLI module. Amongst these, the atomic proposition collector (APC) is particularly important in this context. The breakpoint predicate (*pred*) needs to be interpreted in the subject language, to achieve this the APC function simply projects the *predicate* part of the action pair (`λ c a, a.1`).

The *Finder Bridge* component can be specified through the modular composition of STRs, as illustrated in Figure 2.8. This specification uses the *synchronous_product* operator, defined in Listing 2.8, to compose the subject language STR with the breakpoint semantics, itself captured by the *predicate_asserter* STR (Listing 2.18). The subject language STR needs to be specialized to the *start configurations*. This is achieved by defining a new STR operator, named *preinitialized*, which replaces the *initial* slot of the subject language STR with the *start configurations* set. The breakpoint set specializes the *predicate_asserter* STR. Besides the two STR, the *synchronous_product* operator requires two matching SLI functions. On one hand, the subject language should provide an evaluation function for the diagnosis language, its APE function. On the

other hand, the *predicate_asserter* SLI delegates the evaluation of the breakpoint predicates to the subject language. Hence, it should contribute its APC function. The *synchronous_product* returns a STR, which is mapped to a transition relation through the *STR2TR* operator, defined in 2.10. The *STR2TR* operator requires a predicate capturing the set of accepting states of the transition relation. In our case, this set is induced by the breakpoint semantics on the composite configurations obtained through the product. All configurations for which the *predicate_asserter* slot is true are accepting. This statement is formalized by λ `c, c.2`, which projects the *predicate_asserter* slot of the composite configuration c.

2.4.3 Some Species from the Debugging Zoo

Based on the existence and the structure of the *history* slot in the debugger configuration (*DebugConfig*), different debugging strategies can be achieved:

- *Traceless debugging* [114] is probably the most common case, where the debugger configuration does not have an explicit *history* slot.
- *Omniscient debugging* [76], which stores the history as a linear path from the initial configuration, thus enabling back-in-time jumps.
- *Multiverse debugging* [81], which stores the history as a tree of potential execution paths. This approach, highly effective for non-deterministic languages, allows jumps between different execution histories of the system (universes).

The G∀min∃ debugger, presented in this section, focuses especially on *Multiverse debugging*. Nevertheless, by relaxing and/or strengthening the constraints on the history representation and structure both the traceless and omniscient debugging strategies can be realized. In the case of the traceless debugging, the *history* slot could be simply removed from the *DebugConfig*, which in turn would imply that the jump actions are never enabled. For omniscient debugging a linear trace structure should be enforced, which, amongst others, imply that the future should be erased if the user changes the universe (takes a different path) after a back-in-time jump.

Remote Debugging is a strategy that decouples the debugger itself from the subject language and distributes them over the network. This approach can be made possible by completing the SLI with serialization/deserialization functions. Based on these functions the remote communication can be established between the subject language SLI and the debugger SLI. Furthermore, we believe that our formal debugger definition can be further generalized to a generic debugger specification, which can serve as a basis for proving the functional correctness of arbitrary debugger implementations. To illustrate this point let us consider *tdbg* an *Traceless debugger* implementation, which simply drops the *history* slot from the configuration. To show that *tdbg* satisfies our specification, we need to show that the state-space induced by the *tdbg* implementation simulates the state-space of the specification. Conceptually, this fact can be simply established by introducing a history variable to establish a refinement mapping between the two [105]. The auxiliary variable reconstructs the trace slot contents by capturing the configurations encountered during a *tdbg* session. In our context, this can be realized by simply wrapping the *tdbg* STR with a "history" STR that adds the *history* slot to its configuration. A problem we encounter is that the specification of the *actions* function (Listing 2.16) is too restrictive. It generates one jump action for each configuration in the *history*. To relax this constraint, we should generalize the *actions* function so that it generates jump actions for partitions of the *history*. This way we obtain the empty partition case, which does not enable any jump actions exactly like *tdbg*. With this generalization, it is now possible to provide the refinement mapping, thus prove the simulation relation.

By the way, wrapping the *tdbg* STR with the one defining the *history* variable illustrates once more the composability of our approach. Moreover, without much effort, we can extend this setup further so that the resulting STR offers a modular implementation equivalent to the specification in Listing 2.14. The only requirement is that the *tdbg* SLI should offer an update API, that enables setting the current configuration of the *tdbg* configuration to the one specified by the jump action.

2.5 Scheduling in a Modular Architecture for Verification and Execution

In language engineering, the creation of a new concurrent language (e.g., a Domain-Specific Language (DSL)) usually goes with the design of an execution engine along with different add-ons for model analysis. These tools are useful to execute models (or programs) conforming to this language and formally verify their behavior. During this process, language engineers must handle concurrency by scheduling and synchronizing the execution of the different high-level processes (e.g., threads, actors, active objects) of a model. In particular, "scheduling" solves the non-determinism introduced by the mismatch between the model concurrency and the available physical resources (processors). In general, the scheduler maps concurrent high-level processes in space (on the physical resources available) and in time in adequacy with an optimization objective, known as a scheduling policy.

In the literature, a lot of different mechanisms have been defined to handle concurrent languages (e.g., in SCADE [115], UML [62], Erlang [116]). Using these mechanisms, the scheduler can be either defined in the design model as a separate execution unit or implicitly encoded in the language semantics before being mapped to concepts of the execution platform (e.g., an embedded target, or an operating system). On the one hand, these solutions are not always applicable because of the limited expressivity of the modeling language, or the complexity of transformations (e.g., code generation). On the other hand, scheduling is **not handled at the language-level** but either at the application-level or at the platform-level (i.e., the level of the execution support), which makes it difficult to respect the language atomicity (e.g., atomic assignment of a variable). The scheduler must preserve the language atomicity otherwise its execution may happen while the runtime data used for execution are inconsistent. This may cause the scheduler to choose a step that is not executable according to the language semantics or to make inappropriate choices, which may lead to runtime failures or at least to incorrect behavior of the system.

Therefore, we need a reusable solution that defines an explicit semantic-aware scheduler (by "semantic-aware", we mean that it does not break the language atomicity) and that can be used outside the common thread-based environment.

For verification purposes, concurrency mechanisms should considered in the verification process because they impact model execution and may be responsible for concurrency bugs (e.g., deadlocks). In practice, this consideration remains complex to achieve. To perform model-checking, classical verification approaches need either to use a transformation towards a formal language (e.g., PROMELA for the SPIN [117] model-checker) or to define an abstraction of the concurrency mechanisms (e.g., for the Divine [118] model-checker). Proving that the model transformation is correct or that the considered abstraction conforms to the actual concurrency mechanisms remains a difficult task. Therefore, these techniques still face an equivalence problem between what is verified and what is executed at runtime.

Thus, the design of an explicit and reusable scheduler, that can be used both for deployment on actual systems and for model verification, remains a scientific challenge. To sum up our observations, the complexity of this problem lies in two main factors: *(1)* Concurrency is usually implicitly encoded within the language semantics while the scheduler is defined at application-level or at platform-level, which renders the scheduler hard to tailor to language needs. *(2)* Given the fact that scheduling has an impact on the model behavior, it should be taken into account for the verification phase, but this remains a difficult task.

To address these issues, we introduce a software architecture that enables a modular composition between the system, the scheduler, and the system environment. To illustrate our approach, Figure 2.9 presents the main principles of our architecture using three variants. In all these variants, the *model execution* component represents the execution of the system model composed

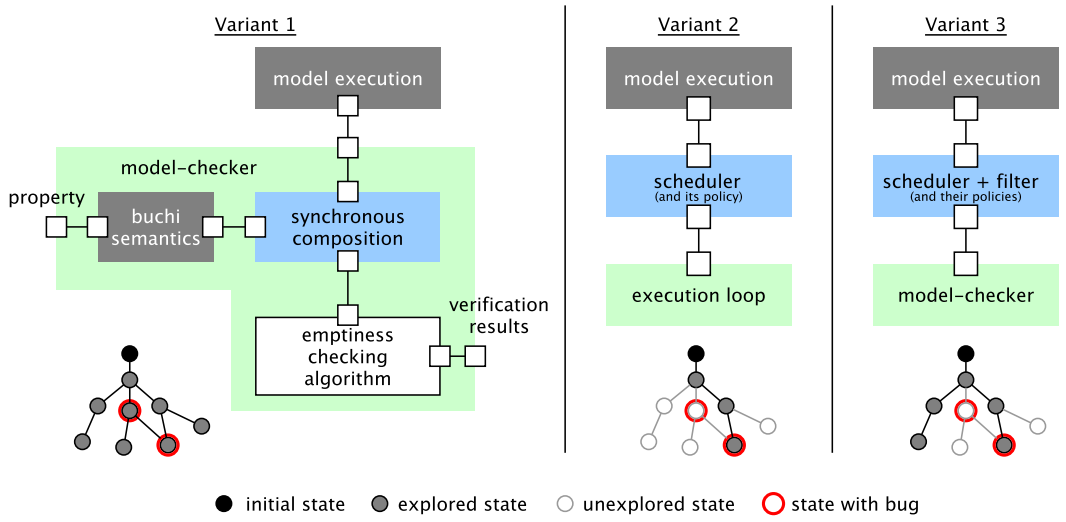


Figure 2.9: Approach overview.

(asynchronously) with the environment. The asynchronous composition with the environment is made either explicitly, through an operator, or implicitly by the model execution engine. While for the execution the real environment is considered, for the verification step, an abstract environment model needs to be constructed [25].

The *model execution* component exposes the list of available execution steps through the G∀min∃ SLI, more precisely through the STR interface. This interface enables the control of the model execution using the modeling language semantics implemented by the engine. Using the STR interface, different operators and/or tools can be combined to control model execution according to the required setup, i.e. runtime execution or formal verification.

Model-checking. For model-checking we follow the modular approach introduced in [14, 1]. The *model-checker* (the green shaded box in the leftmost diagram of Figure 2.9) relies on the STR interface exposed by the *model execution* for the verification of formal properties. The model execution STR is synchronously composed with the property (itself an STR, encoding a buchi automata semantics). The *emptiness checking algorithm* consumes the composition results through a *Transition Relation* interface obtained from the SLI, as presented in subsection 2.3.5 [22].

The corresponding state-space[3] is shown in Figure 2.9 with the initial state in black, explored states in gray, unexplored states in white, and states with bugs or design errors with a red border. In this case, all states have been explored and two errors have been detected.

Scheduled Execution When deploying this model on its actual execution platform (*Variant 2* in Figure 2.9), a scheduler is required to solve the non-determinism induced by the fact that several processes are concurrent for their execution. The *scheduler* becomes here an explicit and controllable component, which can be configured by a scheduling policy to select the next execution step in a concurrent software application. Its execution is triggered by the main *execution loop* of the platform on which the execution engine is running. In this case, the

[3]Representation of state-spaces in Figure 2.9 have been designed for illustration purposes only.

execution only follows one execution trace, and its state-space shows that only one of the two errors is encountered.

Model-checking with Scheduling For the verification phase, the *Variant 3* in Figure 2.9 introduces a set of operators (*scheduler + filter*) in between the *model execution* component and the *model-checker* component (described for variant 1).

The *scheduler*, identical to the one used at runtime, is considered in the verification loop to schedule the execution steps of the system (without interfering with the non-determinism coming from the environment abstraction). Furthermore, we introduce a *filter* component that can encode some high-level hypotheses usually considered in model-checking (e.g., to remove some interleavings). In this case, some execution paths that are impossible, due to scheduling choices, are not explored. The explored state-space still contains several traces because of the non-determinism of the abstract environment. We can also notice that the number of errors in the explored state-space is reduced, and the remaining ones are more likely the ones towards which we are steered by the scheduler.

Our efforts focus on bridging the gap between embedded execution and formal verification of critical systems. In this context, it is of paramount importance to ensure that the real execution is faithful to the formal verification results. Taking scheduler semantics into account during verification helps achieve a high-level of trust in the embedded execution enabling to discover fairness violations (unfair schedulers) while eliminating false requirement violations (scheduling abstractions).

The main contributions described here are **a formally defined, modular and compositional approach for the integration of scheduling in the verification process**. This approach enables, without restrictions, the integration of:

- *arbitrarily complex* modeling languages,
- of scheduling algorithms, *without any restriction*, which can, be deployed in production
- of off-the-shelf, state-of-the-art model-checking algorithms [22].

To the best of our knowledge, this is the first proposal to achieving these goals. Furthermore, the explicit consideration of the interaction between scheduling and the language runtime enables a new class of language-aware scheduling algorithms. These schedulers can guarantee language semantics preservation, much like cooperative scheduling, without, however, requiring trusted software developers but only trusted language runtimes.

Our approach has been applied to the scheduling of embedded systems described as UML [62] models. For our experiments, we use an embedded model interpreter called EMI [1] to execute and verify the UML model of a level-crossing controller *under different scheduling policies*. The UML language is an appropriate language for this experiment because, during the execution of a UML model, different active objects are executed concurrently and should be scheduled. Our study focuses on bare-metal deployment, which, without OS services, cannot rely on POSIX (Portable Operating System Interface Unix) threads. As a result, for runtime execution, our scheduling operator is connected to the interpreter running the level-crossing model. All this setup (system model + interpreter + scheduler) is then deployed on a STM32 discovery embedded board. For formal verification, the scheduled system is asynchronously composed with an abstraction of the environment of the level-crossing system. This software architecture is then controlled by OBP2 [19, 14] (`http://www.obpcdl.org/`), an off-the-shelf explicit-state LTL model-checker.

The remainder of this section is structured as follows. In subsection 2.5.1 we present the classical approaches used for executing and verifying concurrent models. The main architectural

contributions are presented in subsection 2.5.2. In subsection 2.5.3 the approach is instantiated in the context of UML. We discuss some points about our work in subsection 2.5.4. We review some related work in subsection 2.5.5.

2.5.1 Background and Classical Solutions

Here we consider the design of a concurrent language along with its runtime and verification environments. By "concurrent language", we mean a language having several non-exclusive high-level processes available for execution at a given point in time. If their number exceeds the number of processors on the execution platform, these high-level processes are competing for their execution. This kind of execution is called "concurrent execution". In this section, we present the classical approaches used for executing and formally verifying models conforming to a given concurrent language.

Runtime Execution. For the actual execution, two main approaches are usually used: translational techniques and operational techniques. The translational approach makes a mapping between the model of concurrency of the design language and the one of the underlying platforms (e.g., an Operating System (OS), a Virtual Machine (VM)). Different techniques can be used for this purpose including code generation (e.g., Papyrus Software Designer [119], Rhapsody [120]) or compilation to low-level code (e.g., Unicomp [121], GUML[122]). This mapping between a high-level concurrent semantics and the underlying execution platform can be very complex. Different concerns should be taken into account (e.g., scheduling, synchronization mechanisms). All these concerns may be interdependent, which renders this operation even more difficult. Here, we focus on: the model of concurrency, the Inter-Process Communication (IPC) mechanisms, and the scheduling. The high-level processes should be mapped to low-level tasks of the underlying platform. These high-level processes have different names (e.g., tasks, threads, actors, active objects) depending on the high-level language used. In the following, the term Execution Units (EU) is used as a unification concept for all kinds of concurrent processes. Please note that EU only unifies the terminology, but the different concurrency semantics are preserved.

In the same way, the scheduler and IPC mechanisms of the high-level languages should be mapped to the concepts of the underlying platform (e.g., the OS scheduler and message queues for an OS platform). If preemptive, the underlying platform scheduler can break the "atomicity-level" required by the high-level semantics. This can introduce subtle concurrency bugs (e.g., assignment not atomic even if atomic in the source semantics). If cooperative, the underlying platform scheduler needs to trust the application developer that the model will cooperatively release control, i.e. the model will not execute infinitely without returning to the scheduler. Moreover, the scheduler is usually a generic component, which is difficult to align with the language or domain needs.

The operational approach aims at encoding the semantics of the high-level language in an interpreter. In the software language engineering (SLE) community, several model interpreters have been defined. For instance, for fUML [62], the foundational subset of UML, multiples model interpreters are available: Moka [123], Moliz [124] or the reference implementation (`http://modeldriven.github.io/fUML-Reference-Implementation/`). This approach does not suffer from the mapping issues of the translational approach but the scheduling remains quite often implicitly encoded in the language semantics.

For more examples using these approaches, the work in [125] provides a systematic review on model execution in the context of UML.

Formal Verification. For formal verification, three classical approaches are usually considered. A first technique is to use a transformation from the design model to an analysis model usually written in a formal language. As shown in [14], this technique creates a semantic gap between

the design model and the analysis model, which renders the understanding of verification results more difficult. Moreover, an equivalence relation should be built, proven, and maintained to ensure that what is executed at runtime is really what has been verified. This relation remains complex to establish and prove in the general case.

To avoid building such an equivalence relation, another technique consists in proving the correctness of the system running on the underlying execution platform. However, formal verification of the deployed system requires an abstraction of the underlying platform services. In particular, this includes defining an abstraction of the model of concurrency, of IPC mechanisms, and of the scheduler. For instance, execution units of the design language can be mapped to C or C++ processes and formally verified using the Divine model-checker [118, 126, 127] (`https://divine.fi.muni.cz/`). Divine relies on DiOS [126], which offers a "model-checking-friendly" abstraction of the POSIX interface. Relying on an abstraction of the underlying platform services to perform formal verification has several drawbacks. First, the mapping between the design language and the underlying platform concepts creates a semantic gap. As a result, it is more difficult to be sure that the analysis model respects the high-level semantics of the design language. Second, a proof must be provided to ensure that the abstraction of the platform services faithfully implements the *desired* execution platform behavior.

To avoid mapping issues, a third technique based on operational semantics aims at using a modified version of the execution engine for model analysis. For instance, Java PathFinder [128] uses a custom-made Java Virtual Machine (JVM) to perform model-checking on Java bytecode. However, this technique also requires abstract concurrency mechanisms and in particular scheduling. Therefore, to trust verification results, a proof has to be established that these abstractions conform to the real JVM behavior.

To sum up, some issues remain for scheduling and managing concurrency of high-level languages, especially to execute and verify models conforming to these languages in a unified way. To the best of our knowledge, our approach is the first to address the scheduling problem at both runtime and verification levels. Furthermore, our proposition is unique because it introduces a modular approach to scheduling in the verification phase while enabling the deployment of a semantic-aware scheduler on the execution platform.

2.5.2 Architecture for Verification and Runtime Execution

Using the operators defined in section 2.3, we define an executable and verifiable composition between the language semantics, the scheduling, and the environment. At runtime, these operators are used to schedule the execution of concurrent systems running on embedded targets. During the verification phase, the same operators can also be used to consider the scheduler in the verification loop and thus reduce the gap with the runtime execution.

Runtime Execution

To execute a concurrent model, we introduce, in Figure 2.10, the software architecture of an execution engine for a high-level language. The components SEU_1 to SEU_N are the different System Execution Units (SEU) of the executed model. Their execution follows the *language semantics* that encodes the meaning of each concept of the design language. We suggest encoding the language semantics into an interpreter, an operational semantics, that implements the STR interface. Even if generated code can also implement the STR interface, the operational approach has been chosen because it avoids the semantic gap induced by the mapping operation (e.g., code generation) of translational approaches (subsection 2.5.1). The semantics implementation can communicate with the *concrete environment* (i.e., the actual environment) through an Applica-

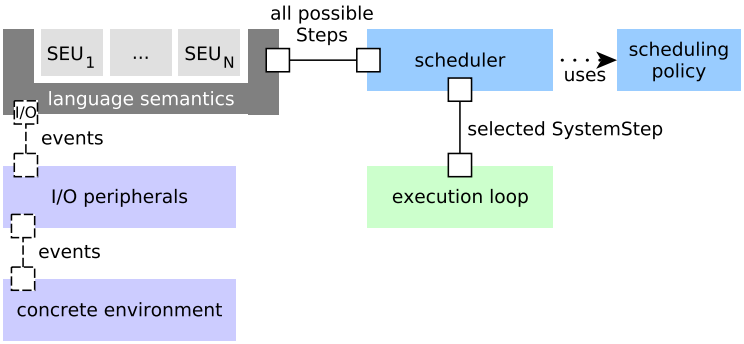


Figure 2.10: Architecture for scheduling the runtime execution.

tion Programming Interface (API) that enables the exchange of data with *I/O peripherals* (e.g., sensors, actuators) of the hardware platform (e.g., an embedded board).

This architecture uses our scheduling operator (*scheduler*), to schedule the concurrent model execution according to a *scheduling policy*. The *scheduler* is controlled by the main *execution loop* of the execution engine that only executes two instructions in a loop. *(1)* The *execution loop* asks the *scheduler* to select the next execution step (*selected SystemStep*) among the set of available actions (*all possible steps*) returned by the *language semantics*. *(2)*[4] The execution of this step is then triggered by the *execution loop*, which delegates its execution to the *language semantics* implementation through the whole architecture. At the same time, the execution state of the *scheduling policy* is updated according to the executed step.

The *execution loop* controls the resulting STR given by the following formal definition of our software architecture.

Definition 2.5.1. The software architecture for runtime execution is formally defined in the following way:

```
def runtime_execution
    (system : STR C A)
    (scheduling_policy : SchedulingPolicy C A S)
: STR (C × S) ( S × A) := scheduler C A S system scheduling_policy
```

The *scheduler* can be configured with different scheduling policies gathered into two categories. *Stateless scheduling* policies can choose an execution step only based on the set of possible execution steps and the content of the current configuration. For instance, the fixed priority scheduling policy is an example of a stateless scheduling policy. It selects the execution step that belongs to the execution unit with the higher static priority. In contrast, *stateful scheduling* policies also require an execution state to store some persistent data. As an example, round-robin is a stateful scheduling policy that selects an execution step of each execution unit in turn. For this purpose, it requires storing the *turn* variable that indicates which execution unit has the highest priority at the next execution loop.

Using this *scheduler* component has several benefits. First, we have a semantic-aware scheduler, which is independent of the used model of concurrency but bound to the language semantics

[4]The data exchanged during the execution of a step is not displayed in Figure 2.10 but it uses the same links and ports as *(1)* in charge of computing the next execution step. This is also the case for all figures in this document.

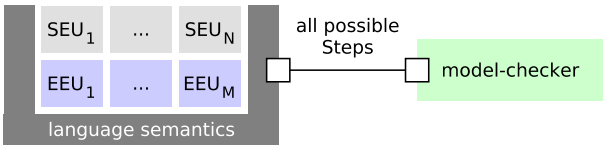


Figure 2.11: Architecture for model-checking.

through the STR interface. The scheduler appears explicitly in our architecture and is configured by a scheduling policy to adapt to the application or domain needs.

Second, this scheduler respects the semantics of the language such that it cannot break the language atomicity. For instance, for UML, it will respect the Run-to-Completion step semantics, which assumes that the next event cannot be dispatched until the processing of the last event has finished. The scheduler also must trust that the language semantics will release control and not execute infinitely. The trust is delegated to the language designer (i.e., the engineer in charge of the language semantics implementation), who is supposed to know what to do, and not to the application developer or to the underlying platform developer (as is usually the case). Third, in contrast to RTOS schedulers that mix both the scheduler and the main execution loop, we decouple these two components to make the *scheduler* a reusable and controllable component for the verification phase.

Initial Model-checking Approach

Figure 2.11 shows how model-checking is usually applied to the design model when an execution engine for the design language is available, as in [14]. In contrast to runtime execution, the system must be closed by an abstraction of the actual environment for formal verification. This abstraction is modeled as different concurrent Environment Execution Units (EEUs), which are executed by the same *language semantics* component. These execution units are asynchronously composed with the system execution units. The *model-checker* is here directly connected to the *language semantics* implementation. For each configuration returned by the execution engine, it gets all possible execution steps. As a result, the model-checker explores all possible execution paths and gets the whole model state-space. However, in industrial systems, this can lead to state-space explosion especially due to the interleaving of events coming from the environment. To avoid the state-space explosion, some approaches aim at using compositional model checking [129] or smart environment generations [130] but it may also require considering some high-level hypotheses.

Model-checking with Filtering

Introducing the *reactive hypothesis* is one strategy used to reduce the interleaving of events coming from the environment. This hypothesis, initially considered for synchronous languages [131], assumes that the system execution is infinitely faster than the environment reactions. This means that if the system has some possible execution steps, one of them must be executed, else an execution step of the environment has to be taken. Therefore, the response of the system to any given environment stimuli is not interleaved with other environment events.

From a practical point of view, the reactive hypothesis acts as a filter on available actions. The software architecture presented in Figure 2.12 can be used to consider the reactive hypothesis for model-checking purposes. In Figure 2.12, our *filter* operator is introduced in between the *language semantics* implementation and the *model-checker*. This *filter* is configured by a *filtering policy*

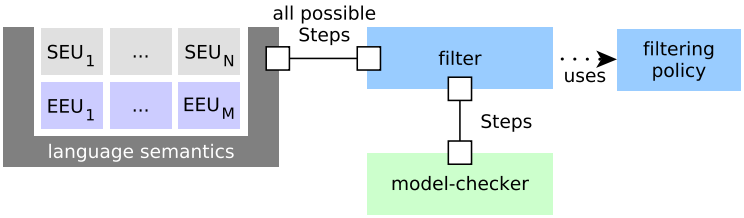


Figure 2.12: Architecture for model-checking with filtering.

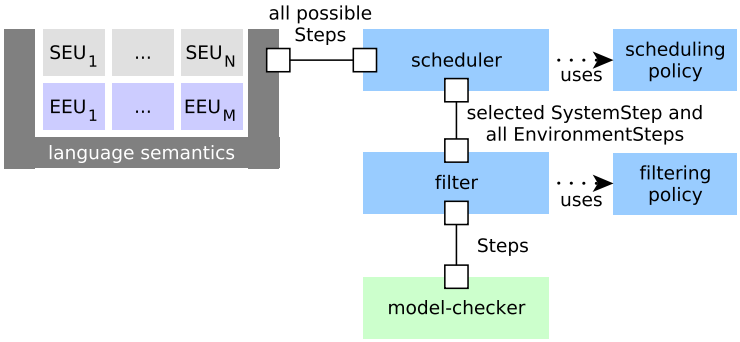


Figure 2.13: Architecture for model-checking with the scheduler in the verification loop.

that specifies which execution steps should be filtered. For instance, to enforce the reactive hypothesis, the *filtering policy* has to select an execution step of the system or, if none exists, an execution step of the environment. The reactive hypothesis is an instance of the filtering policy. Nevertheless, the architecture presented is more general, allowing any filtering policy. For instance, another filtering policy might be used to remove actions that cause overflow on fixed-size message queues.

Definition 2.5.2. The model-checking architecture with filtering is formally defined in the following way:

```
def model_checking_with_filtering
    (sys_and_env : STR C A)
    (filtering_policy : FilteringPolicy C A S)
: STR (C × S) (S × A) :=
filter C A S sys_and_env filtering_policy
```

In all formal definitions until the end of the section, these definitions give the resulting STR that is controlled by the *model-checker*.

Model-checking with the Scheduler in the Verification Loop

This section describes how to take into account scheduling in the verification loop. This is one of the main novelties introduced in this manuscript. Considering such a platform-specific aspect during the model-checking phase enables to remove some execution paths, from the model state-space, that are never executed at runtime due to scheduling choices.

As mentioned in section 2.5.2, an abstraction of the real environment is required to perform formal verification. The initial reflex to include the scheduler in the verification loop would be to

manually extend the scheduling policy so that the scheduler can choose arbitrarily among both the system and the environment execution steps. However, this simplistic approach introduces an environment-abstraction-induced interference in the scheduling because this provides an ordering of the system execution steps potentially different from the actual runtime execution ordering.

A better solution is simply to exclude the environment execution steps from the scheduling choice. We do not want to schedule the environment especially because of its intrinsic non-determinism and because we usually do not know how to schedule environment steps (i.e., according to which scheduling policy?). This is the reason why the scheduler has simply to forward all environment execution steps and chooses one from the system execution steps. This solution is described in Figure 2.13. This software architecture presents how to compose the system, the scheduler, and the environment abstraction in a modular way for model-checking purposes.

This architecture relies on our scheduling operator. Its `selector` function enables the scheduling only on steps coming from SEUs. All possible system steps are then processed by the *scheduler*, which selects one of them according to the chosen *scheduling policy*. The *selected SystemStep* is then merged with all possible *EnvironmentSteps* before being sent to the *model-checker* either directly or via a *filter* component (e.g., to enforce the reactive hypothesis).

Definition 2.5.3. The model-checking architecture with the scheduler in the loop is formally defined in the following way:

```
def model_checking_with_scheduling
    (S₁ S₂ : Type)
    (sys_and_env : STR C A)
    (scheduling_policy : SchedulingPolicy C A S₁)
    (filtering_policy : FilteringPolicy
        (C×S₁) (S₁×A) S₂)
: STR ((C×S₁)×S₂) (S₂×(S₁×A)) :=
filter (C×S₁) (S₁×A) S₂
    (scheduler C A S₁ sys_and_env scheduling_policy)
    filtering_policy
```

To ensure reproducible executions in model-checking, the scheduling policy should be deterministic i.e., given a set of possible execution steps, a given configuration, and a given scheduling state, the scheduler must always select the same execution step. This does not contradict the fact that model execution can have non-determinism. However, a deterministic scheduler is required, for model-checking, to solve the system non-determinism while forwarding the environment non-determinism to the model-checker. Even a random scheduling policy is deterministic if the state of the random-number generator algorithm is saved in the execution state of the scheduling policy. At each step, this state can be restored such that the random-number generator returns a deterministic number (see *rand_r* function of C programming language).

This approach for model verification offers a modular way to compose the system, the scheduler, and the environment with two operators: a scheduler and a filter (e.g., for reactive hypothesis enforcement).

The software architecture considers the scheduler for the verification phase and thus reduces the gap between runtime execution and model verification. It also improves model-checking performance by avoiding exploring some execution paths, which are impossible due to the choice of the scheduling policy. However, this consideration also has a counterpart. The verification performed is not valid anymore with other kinds of scheduling policies. The execution engine should be deployed with the same scheduling policy as the one used during the verification phase. Otherwise, the correctness of the model behavior would not be ensured. Another limitation is

that the environment execution may interfere with the system execution because both system and environment execution units share the same computing resources. As a result, the environment execution may alter the execution state of the system (e.g., in the case of dangling memory pointers).

Model-checking with decoupled Environment

To address these interference problems, we introduce a second concurrent language semantics (*language semantics (env)*), running on the same processor (i.e., not in a distributed way), that only interprets the environment abstraction. Such decoupling makes it possible to easily connect different environment abstractions to the system execution by only replacing the environment execution runtime. This can also be interesting for isolating each execution in separate memory areas to preserve the integrity of their execution states. This software architecture is illustrated in Figure 2.14. The environment abstraction needs to be composed asynchronously with the scheduled system semantics produced by the *scheduler* to obtain the closed model needed by model-checking algorithms. This task, which cannot be made anymore by the language semantics, is performed by the *asynchronous composition* component that provides an implementation of our asynchronous composition operator. A *filter* can then be used to filter some actions and enforce some hypotheses (e.g., the reactive hypothesis) on the STR provided by the *asynchronous composition*. This setup is finally controlled by a *model-checker* that performs exhaustive verification on the whole model state-space.

Definition 2.5.4. The model-checking architecture with the environment model decoupled from the system model is formally defined in the following way:

```
def model_checking_with_decoupled_env
    (C₁ C₂ A₁ A₂ S₁ S₂ : Type)
    (system : STR C₁ A₁)
    (environment : STR C₂ A₂)
    (scheduling_policy : SchedulingPolicy C₁ A₁ S₁)
    (filtering_policy : FilteringPolicy
            ((C₁×S₁)×C₂) ((S₁×A₁) ⊕ A₂) S₂)
: STR (((C₁×S₁)×C₂)×S₂) (S₂×((S₁×A₁) ⊕ A₂))  :=
filter ((C₁ × S₁) × C₂) ((S₁×A₁) ⊕ A₂) S₂
    ((scheduler C₁ A₁ S₁ system scheduling_policy)
        ⊗ₐ environment)
    filtering_policy
```

Moreover, environment execution units need to exchange data with system execution units. For this purpose, the *I/O* ports of both language semantics are connected to bind the environment outputs to the system inputs and the system outputs to the environment inputs. In our software architecture, the communication is made through shared variables such that runtime data of both execution engines can be linked together. It may be possible to use different IPC mechanisms by defining other composition operators, besides the asynchronous composition.

If both system and environment execution units are specified in the same language, *language semantics (sys)* and *language semantics (env)* can be two instances of the same language semantics implementation.

However, it is also possible to specify environment execution units in a different language thus producing a heterogeneous model execution.

This would be like the Ptolemy directors [132], with the added constraint of defining a data marshaling strategy between the two languages.

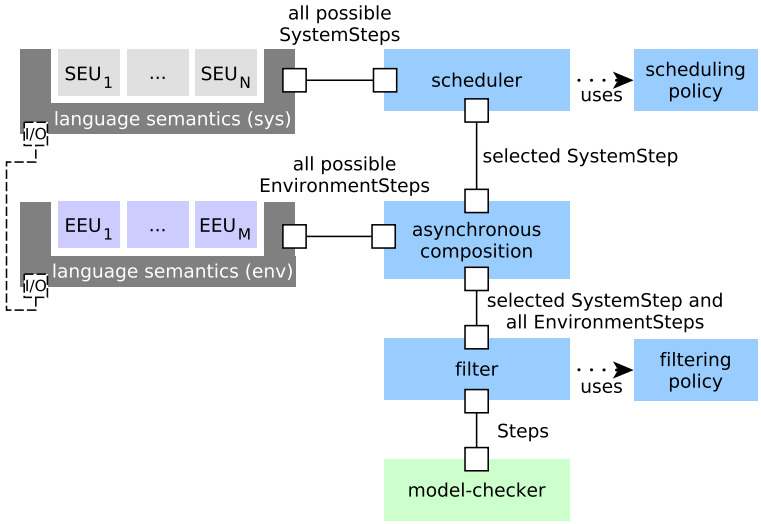


Figure 2.14: Architecture for model-checking with the scheduler in the verification loop and decoupled environment.

To sum up, this work presents a modular software architecture that can be used to compose the system, the scheduling, and the environment for both runtime execution and model-checking. This composition uses three operators to integrate scheduling in the formal verification process.

2.5.3 Illustration on UML

For practical experiments, this approach has been applied, in the context of the PhD of Valentin Besnard, on a UML model interpreter running a model of a level-crossing controller. These experiments aim at evaluating our approach for the development of concurrent models from formal verification to embedded execution.

A Bare-metal Interpreter of UML Models

EMI (Embedded Model Interpreter) is a state-of-the-art model interpreter for UML [62] models (cf. subsection 2.5.5) that was introduced in [28, 133, 1]. The UML subset supported by EMI can be represented by class, composite structure, and state machine diagrams. This execution engine can be deployed on embedded boards (e.g., STM32 discovery) as well as desktop computers. This tool can also be used for simulation and verification with the open-source model-checker OBP2[14]. Furthermore, it also supports runtime monitoring of UML [15].

Given that EMI can run on bare-metal (i.e., without OS), this is an appropriate example where the mapping to the underlying execution platform cannot be made. Indeed, a translational approach (described in subsection 2.5.1) is not possible because EMI is not running on a software underlying platform (e.g., VM, OS) but directly on the microcontroller facilities. As a result, EMI must define its own concurrency and scheduling mechanisms. Three different services are provided by this execution engine regarding concurrency concerns. *(1)* EMI relies on a concurrent model based on green threads to achieve pseudo-parallelism. Indeed, because EMI is usually executed on mono-core microcontrollers, it cannot use real parallelism. Each green thread is running a UML active object with its own state machine as behavior. *(2)* EMI also defines its own IPC

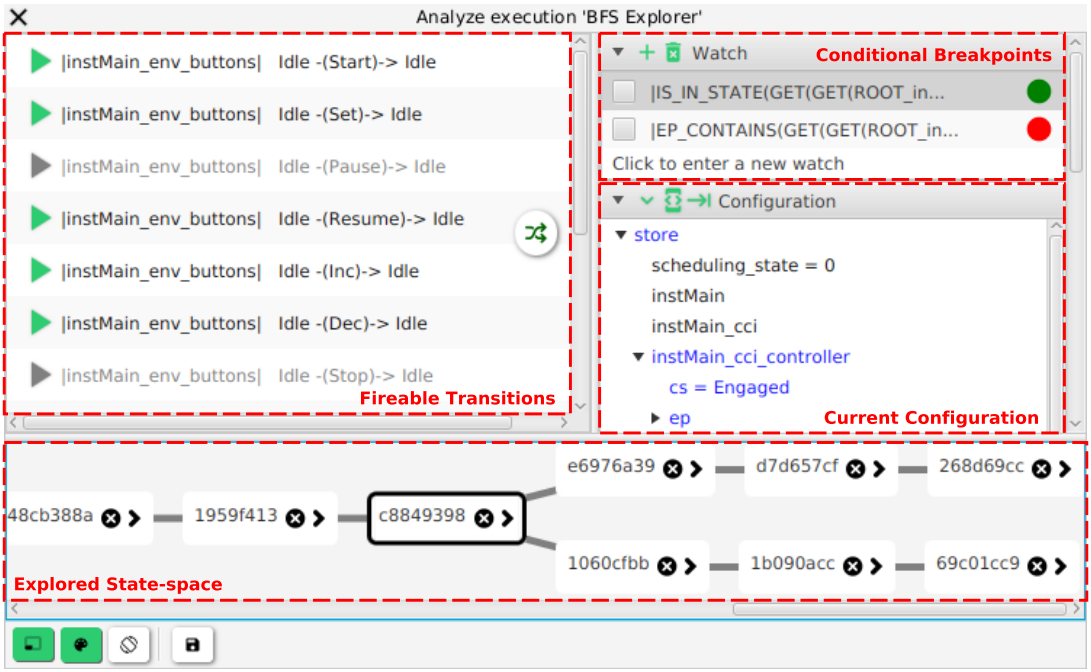


Figure 2.15: OBP2 graphical interface for simulation and multiverse debugging.

mechanisms to exchange data between UML objects. Each object uses UML signal events to send data to another object. These events are stored in the event pool of the receiver object before being processed by its state machine to make progress execution. *(3)* The scheduler is implemented using the scheduler operator, presented here, configured with different scheduling policies (e.g., round-robin, fixed priority scheduling). For both the concurrency model and IPC mechanisms, EMI relies on standard UML concepts for which the behavior is described in the UML specification [62]. However, the UML specification does not cover the scheduling aspect.

In practice, EMI offers an operational semantics implementation of the UML semantics and native implementation of the scheduling semantics. It may also be possible to define an interpreter to execute the scheduling semantics but for performance purposes, the scheduling semantics (i.e., the scheduler and scheduling policies) is here directly described in C, the native implementation language of the tool. EMI is currently conforming to Figure 2.10 for runtime execution and to Figure 2.13 for formal verification. Thus, both the environment abstraction and the system application are executed with the same language semantics. It offers the possibility to put the scheduler in the verification loop and has an implementation of the reactive hypothesis that can be applied as a filtering policy.

Considering the Scheduler during Model Debugging. Before formal verification, an essential step has been to design the UML models for each system. The OBP2 model-checker provides an implementation of the G∀min∃ unified debugger, presented in section 2.4.

In Figure 2.15 we present a screenshot of the debugger UI instantiated on an UML execution setup integrating the scheduler. Since the OBP2 debugger is equally based on the STR interface the addition of the scheduler needed only a decorator for the state-projection function which adds the scheduling-state ($scheduling\_state = 0$ on the right in Figure 2.15) to the current-configuration view.

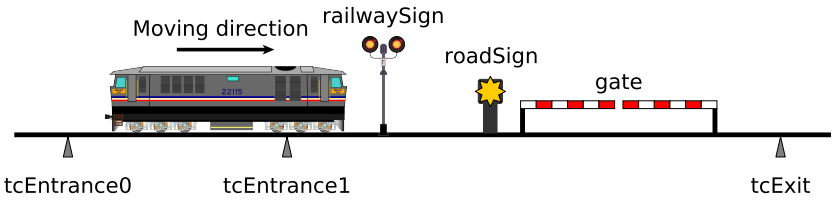


Figure 2.16: Schema of the level-crossing system.

Application to a Case Study

After having described the conceptual solution and its application to EMI, we illustrate our approach by an experiment on a UML model of a level-crossing controller described in [14]. This system aims at protecting the passing of a train at the crossing of a road with the railway. The schema in Figure 2.16 gives a graphical representation of the system. Three sensors (*tcEntrance0*, *tcEntrance1*, *tcExit*) are placed all along the railway to detect the passing of a train. At the arrival of a train, the *roadSign* is switched on and the *gate* is closed. When the level-crossing is ready, the *railwaySign* gives the authorization to the train driver to go on the level-crossing railway section. When the train has left the level-crossing, the *gate* opens and the *roadSign* is switched off. For this experiment, we design a UML model of this system using 8 active objects that communicate together with signals. All state machines of these objects have a total of 15 states, 7 pseudo-states, and 22 transitions.

In [14] the authors show that different event pool implementations can be used to change the event dispatching strategy, which is a semantic variation point in UML. The authors have used a *FifoEventPool* strategy that drops ignored events and an *OrderedListDeferredEventPool* strategy that defers ignored events such that they can be processed later. In our experiment, we reuse these two event dispatching strategies to see if scheduling can have an impact on the model behavior. For this purpose, the two following research questions are used to evaluate our work:

- **RQ1**: Does the approach enable focusing on relevant design errors and improve model-checking efficiency?
- **RQ2**: Does the approach enable helping engineers to choose a scheduling policy and an event pool strategy in adequation with the system correctness?

For these experiments, we use the OBP2 model-checker to explore the state-space of the level-crossing model under different setups, search deadlocks, and verify the same four properties as in [14]:

P1 The *gate* is closed when the *train* is on the level-crossing.

P2 The light of the *roadSign* is active when the *train* is on the level-crossing.

P3 The *gate* finally opens after being closed.

P4 The light of the *roadSign* finally shuts down after being activated.

Our approach does not constrain the use of model-checking tools or the expressivity of properties. In practice, any ω-regular properties can be verified (e.g., liveness or fairness properties) if the model-checker provides such facilities.

Model-checking setup	C	A	D	P1	P2	P3	P4
No scheduling	173	276	2	✓	✓	✓	✗
Round-robin scheduling	26	25	1	✓	✓	✓	✓
Fixed priority scheduling	21	21	0	✓	✓	✓	✓
No scheduling with RH	50	54	2	✓	✓	✓	✗
Round-robin scheduling with RH	8	7	1	✓	✓	✓	✗
Fixed priority scheduling with RH	21	21	0	✓	✓	✓	✓

(a) With *FifoEventPool* strategy

Model-checking setup	C	A	D	P1	P2	P3	P4
No scheduling	122	209	0	✓	✓	✓	✓
Round-robin scheduling	28	28	0	✓	✓	✓	✓
Fixed priority scheduling	21	21	0	✓	✓	✓	✓
No scheduling with RH	32	48	0	✓	✓	✓	✓
Round-robin scheduling with RH	27	27	0	✓	✓	✓	✓
Fixed priority scheduling with RH	21	21	0	✓	✓	✓	✓

(b) With *OrderedListDeferredEventPool* strategy

Table 2.1: Model-checking results with different event pool strategies (with C: the number of configurations, A: the number of transitions, D: the number of deadlocks, and property verification results).

For formal verification, we consider three different cases of scheduling: "No scheduling" when the scheduler is not used, "Round-robin scheduling" when the scheduler is used with a round-robin policy, and "Fixed priority scheduling" when the scheduler is used with a fixed priority policy. We also want to check the influence of the Reactive Hypothesis (RH) on this case study.

By running a Depth-First Search (DFS) algorithm for state-space exploration, a reachability algorithm for deadlock detection, and the "nested DFS" algorithm [134] for LTL model-checking, we collect various results with the OBP2 model-checker on the model executed by EMI. The Table 2.1a shows the results of our experiments with the *FifoEventPool* strategy while Table 2.1b sums up results with the *OrderedListDeferredEventPool* strategy. For each setup, these tables give the number of configurations (C) and the number of transitions (A) of the model state-space, the number of deadlocks (D), and the result of each property verification. Among these results, we can notice that the first line of each table gives identical results as in [14] because the same setup is considered. These results also show that including the scheduler in the verification loop helps to reduce the state-space, tends to reduce the number of deadlocks, and tends to remove some impossible execution-paths that cause property P4 to fail (RQ1). As expected, the use of the reactive hypothesis reduces the state-space size because it decreases the number of interleavings from events coming from the environment (RQ1). These trends have been confirmed by applying the approach to different UML models. In general, the model state-space is reduced because the non-determinism of the system execution is solved. However, the scheduling execution state can differentiate otherwise equivalent configurations.

In the case of this specific level-crossing model, we can observe that the fixed priority scheduling gives better results than the round-robin scheduling (RQ2). This may be explained by the fact that the fixed priority scheduling enables to better customize priorities among all UML active objects. We can also notice that the case with "Round robin scheduling with RH" for the *FifoEventPool* strategy has very few states in its model state-space. In fact, the nominal scenario results in a deadlock, which blocks model execution and causes P4 to be violated. Without the

reactive hypothesis ("Round robin scheduling"), a similar issue is observed except that the deadlock that causes P4 to be violated has been avoided but another deadlock is still there. From our perspective, it is very good for engineers to notice this issue as soon as the verification phase (RQ2). To solve the problem, engineers can either change the event dispatching strategy, the scheduling policy, or both. Moreover, all properties that were verified without scheduling are also verified with both kinds of scheduling policies.

To sum up, we believe these results bring benefits and interesting feedback for engineers. Introducing the scheduler in the verification loop ensures that it does not interfere with the behavior of the system, if a non-interference proof is missing. Moreover, considering the scheduler renders the verification procedure more efficient (RQ1). Furthermore, these results bring interesting feedback for engineers to ensure that the right scheduling policy has been chosen and that the semantic variation points have been appropriately configured (RQ2).

2.5.4 Discussion

This section emphasizes the strengths of our approach, discusses the overhead, and addresses some current limitations, which points to the need for further research efforts.

Strengths

Our approach hides the language semantics behind the STR interface; thus, the model-checking core is isolated from the semantics. The OBP2 model-checker is language agnostic, needing only the STR interface abstraction. Our architecture uses a composable approach that encapsulates the model-scheduling semantics behind the STR abstraction, which is consumed by the model-checker. If the language semantics can be defined through the STR interface, arbitrarily complex concurrency models can be realized. EMI implements the UML semantics in a bare-metal interpreter, its complexity showcases this point. Another strength of our approach is the support for composing heterogeneous semantics. Composing the EMI-UML interpreter with a scheduler implemented in C illustrates this point.

It is important to note that our solution does not alternate between the scheduler and the model steps. The scheduler operates as an inline filter to choose the executable steps during the execution, which executes the model step updating the scheduler state. This guarantees atomicity, a strength of our approach.

Second, our approach has been illustrated in the execution of a UML level-crossing model running on EMI. However, this approach can also be applied to other concurrent languages. For this purpose, the execution engine of a language must fulfill two conditions. *(i)* The tool needs to implement the STR interface such that it can be connected to our operators. *(ii)* To separate actions of the system from those of the environment and to define scheduling policies, the execution engine must provide some introspection capabilities about the content of configurations and the meaning of transitions. Under these conditions, any execution engine can replicate our software architecture.

Overhead

By integrating the scheduling our approach can reduce the overall complexity of the verification problem. However, it needs to be supported by 3 "components": the model executor, the scheduler, and the model-checker. While this approach incurs a high initial tool development cost, due to its composability, it has an overall systemic benefit, by isolating the problems and ultimately enabling reuse. The use of EMI-UML language and OBP as off-the-shelf components proves this point.

Furthermore, the model-checking proves that the system-scheduler composition satisfies the specification. Consequently, it is safe to deploy the system, along with the scheduler, without modification, which alleviates the need to establish an equivalence relation between the verification model and the real execution (which would be necessary in the case of abstract specifications). This point is proven by our experiment, which shows that it is possible to deploy the model running on the EMI-UML interpreter and the C schedulers on an STM32 target. Nevertheless, it should be noted that in this study we do not address the scalability of model-checking in the context of executable specifications. However, we strongly believe that our approach could be used in combination with automated model abstraction verification setups, such as CEGAR [135].

Quantitatively our approach does not add additional state-variables, besides the state of the scheduler, which is also present in other approaches (Section 2.5.5), which consider the scheduler as a "part" of the model.

Current Limitations

Our approach introduces a semantic-aware scheduler. However, if the language atomicity is guaranteed at the language level, the system model must ensure that it releases control. Related to this last point, the use of UML models can still lead to some issues. Indeed, in the action language used by EMI, nothing prevents the model designer from adding an infinite loop in an effect of a UML state machine transition, which breaks the assurance that the execution of a step will always return.

This approach is applied to scheduling policies in the context of embedded critical systems, such as those integrated into real-time operating systems. However, the timing and schedulability analysis are seen as orthogonal issues outside the scope of this study, which focuses on the integration of schedulers in the verification environment.

The environment can be composed with the system using a specialized composition operator, like Ptolemy directors. This aspect is partially covered in [25]. Nevertheless, running the system and the environment in a decoupled and heterogeneous way is a challenge out of the scope of this study. We mention this idea in relation to Figure 2.14 where the system and the environment can be executed with different language semantics. A heterogeneous execution induces several other issues like the synchronization and the coordination of both system and environment executions. Several works in the model-driven engineering community focus on these issues. A dedicated coordination language like BCOol [136] may help to solve this task.

2.5.5 Related Work

This work focuses on executing and verifying concurrent models using a modular architecture, nevertheless the literature is rich with proposals addressing similar problems. In the following we overview some of them. To handle concurrency, various mechanisms have been shown to be quite efficient in practice, while being completely different from standard OS mechanisms. The plethora of libraries, languages, and concurrency models available, show the lack of adequacy between today's OS thread-based concurrency model and the needs of application developers. In the context of critical systems, this is shown by the industrial use of: SCADE [115], Simulink [137], Ptolemy [132], SDL [138], and UML [62] for application development. The microkernel approaches for OS development (e.g., seL4 success [139]) emphasizes the need for modular OS design, which can enable a more natural integration of high-level concurrency models as domain-specific service-libraries. These foster reuse and encourage the development of more appropriate abstractions going beyond the thread-like parallelism models. Furthermore, the research efforts

around CSP/occam [140] or Erlang [116] show that language-specific concurrency mechanisms are more adequate and faster than today's OS services. As a result, all these concurrent mechanisms seem more in line with high-level language needs rather than the thread-based concurrency model of OS.

In the SLE community, the GEMOC Studio [141] provides its own solution for handling the concurrency of executable DSLs. GEMOC Studio is both a language workbench to design new DSLs and a modeling workbench to design, execute, and analyze models conforming to these languages. In [40, 142, 143], the GEMOC team defines a declarative meta-language, called MoCCML, to explicitly specify the concurrency concerns of a DSL at the language level. MoCCML enables specifying the constraints that define how the concurrent control flow of the language will behave, i.e., which execution paths are possible during model execution. In GEMOC Studio, the tool TimeSquare [144] is used to enforce these concurrency constraints and enables to observe the concurrency behavior of a model (e.g., using Visual Constraint Diagrams (VCD)). During model execution, the choice among the remaining valid execution steps is made by the user through the "step decider" addon or by the main execution loop of the simulator. In contrast to GEMOC Studio, our work focuses on defining a modular way to compose the scheduler with the executed model for both embedded execution and model-checking. For implementing the scheduler, our approach enables low-level implementations (e.g., in C) but may also accommodate high-level scheduling DSLs (e.g., MoCCML). In that sense, both works are complementary.

Some related works define explicit schedulers. The work in [145] proposes to slightly modify the fUML [62] execution model to introduce an explicit scheduler with different possible scheduling policies. With these updates, fUML execution engines may capture real-time systems' execution semantics more precisely. In Real-Time OS (RTOS) (e.g., FreeRTOS [146], Trampoline [147]), it is also a common practice to define the scheduler as an explicit component. However, RTOS schedulers usually mix both the scheduling and the main execution loop such that the call to the scheduler is the last instruction of the *main* function. All these projects have explicit schedulers to better capture the language semantics for actual execution, but they do not consider using such components in model-checking activities.

In terms of formal verification, some model-checkers consider some characteristics of the execution environment to improve verification performance. In [148], an extension of Java Pathfinder takes advantage of platform-specific restrictions to reduce memory and time used by state-space exploration. This work suggests bounding the number of threads that can run in parallel to the maximum number of threads supported by the execution platform. Some other works [149, 150] suggest using memory-model aware model-checkers by integrating a formal description of the language memory model (the .NET memory model for [149] and the Java memory model for [150]) into model-checking tools. This approach especially enables detecting further data-races by exploring additional reachable states due to the reordering of some instructions. All these works take into account platform-specific mechanisms to get better verification performance but none of them puts the scheduler in the verification loop.

Related to the lack of modularity and reusability, some tools focus on modular techniques to customize model execution and model analysis. The Bogor framework [151, 152] provides an extensible input language for defining domain-specific constructs and an interface to perform domain-specific optimizations (e.g., on state-space encoding). Some other approaches rely on the template semantics technique to customize the semantics encoded in code generators [153] or in model transformation tools [154, 155] used to obtain analyzable models. In a similar way, the work in [156] can be used to configure semantic variation points of UML-RT models to improve the model-checking efficiency. Polyglot [157] provides a way to execute models with different pluggable semantics for different statechart variants and analyze these models with Java PathFinder. All these works enable to customize the modeling language semantics but

none of them consider the scheduler as a separate and controllable component. In comparison to these works, our approach proposes a compositional approach for designing a scheduling-aware verification environment, which is decoupled from the language semantics and can be reused for different DSLs.

2.5.6 Conclusion

Concurrency concerns are part of the definition of a concurrent language and must be considered both for model execution and formal verification. We have presented an approach to defining a modular composition of the system, scheduling, and environment such that the scheduler, used at runtime, can be integrated into the verification loop.

For runtime execution, the scheduler is controlled by the main execution loop of the execution engine to schedule the system execution. In this case, the interaction with the actual environment is made directly through I/O peripherals of the execution platform.

For formal verification, the scheduler can be integrated into the verification loop to reduce the equivalence problem between what is executed and what is verified. This composition is made with our asynchronous composition operator and can enforce different hypotheses (e.g., the reactive hypothesis). These hypotheses are implemented as filtering policies applying a partial filtering to the available set of actions. To sum up, this approach makes the scheduler an explicit and controllable component that can be easily configured by different scheduling policies. It can be composed with the system and the environment for both execution and verification activities.

Our approach has been applied to a UML model interpreter, called EMI, that can run on embedded targets or be connected to the OBP2 model-checker to detect property violations and concurrency bugs. EMI defines its own model of concurrency based on green threads, its owned IPC mechanisms, and an explicit scheduler. We have illustrated this approach on a level-crossing model by showing interesting results. Integrating the scheduler in the verification loop can help to avoid some bad execution paths that result in property violations or deadlocks. In the verification phase, our approach gives feedback to engineers about the efficiency of a given scheduling policy for a particular model. In general, it also contributes to improving verification performance by reducing the model state-space to explore.

2.6 Conclusion

In this chapter, we have introduced the G∀min∃ SLI. We have shown that this interface is sufficient to implement in a language-agnostic way 5, rather common, operators: *filter*, *scheduler*, *interleaved and synchronous composition*, and a conversion operator, which lower the SLI interface to a transition relation. The strength of the approach was illustrated through: (1) the formal specification of a multiverse debugger, a rather complex composite monitor relying on a model-checker for breakpoint lookup, and (2) the formalization of a novel software architecture, which enables the modular introduction of the runtime scheduler in the verification setup. These formalizations show that the G∀min∃ SLI allows the creation of unique non-trivial analysis tools, of special interest for the diagnosis of executable specifications. The realizability of our approach was illustrated by presenting EMI, a bare-metal interpreter of UML models implementing the G∀min∃ SLI, which, when coupled with the OBP2 tool, offers a prototypical implementation of the formalizations discussed here. The interested reader can refer to:

- [1, 25, 15, 14] for further details related to the EMI UML experiments;

- [82, 17, 16] for further details on multiverse debugging and AnimUML, another UML implementation strongly inspired by our formalization;
- [20, 19] for an in-production implementation of the G∀min∃ SLI along with the model-checking bridge offered by the OBP2 tool (http://www.obpcdl.org).

The following chapter will conclude this manuscript discussing some future research directions enabled by the contributions discussed in this chapter.

Chapter 3

Conclusion & Perspectives

Contents

3.1 Conclusion

In the last ten years, my research was oriented toward understanding the interface between the semantics of executable specification languages and the tools needed for behavioral analysis during the diagnosis process. One of the high-level objectives behind my research efforts is rendering the design of executable specifications as fun and productive as programming in a dynamic language environment while ensuring formal guarantees of the analysis results. Amongst the main challenges, we note the large number of domain-specific executable specification languages, which need behavioral analysis tools. We observe a high reliance on transformation approaches that leads to semantic duplication and brittle architectures, sometimes difficult to maintain. This points to the need for an in-depth architectural rethinking of the connection between the executable-specification languages and the behavior analysis tools.

Our vision is reminiscent of the seminal work [74] of A. Kishon, P. Hudak, and C. Consel that introduced *monitoring semantics*, a model which captures the monitoring activity found in some program analysis tools, such as debuggers, and profilers. However, to obtain reusable monitors we strive for a higher degree of independence between the language semantics and the monitors, which are seen as semantics dependent on the subject language and not only as an extension of the subject language semantics. From this perspective, the G∀min∃ SLI, presented in this manuscript, offers a generic API interface that mediates the exchanges between the language and its monitors. The G∀min∃ approach can be seen as a generalization of the *monitoring semantics* that accommodates well the constraints imposed by the executable specification languages. The versatility of our approach was illustrated, in this manuscript, through the formalization of a multiverse debugger monitor and the formal specification of a novel software architecture that brings the runtime scheduler into the verification loop, without requiring model transformations. The realizability of our vision was sketched through the analysis of an illustrative model executed on the EMI bare metal UML interpreter, which along with the OBP2 tool implements the formalization discussed in chapter 2.

We have significantly improved the state-of-the-art paving the way towards a modular and composable approach for building executable specification analysis tools. Besides the respective scientific papers, our contributions are crystallized in the development of 3 significant open-source research prototypes, amongst which OBP2 – a language agnostic model-checking monitor – is currently commercialized with two products, PROCESS and STUDIO, from PragmaDEV.

We are grateful for the financial support offered by 12 research grants that allowed us to collaborate with numerous researchers, practitioners, postdoctoral fellows, PhD candidates, and research engineers, all of whom significantly contributed to the realization of our scientific vision. The next section offers some insights on a broader vision rich in open research questions, which, hopefully, will fuel the community for the next decade.

3.2 Perspectives

This chapter overviews some perspectives of my work, focusing on the main directions I would like to investigate during the next few years. The discussion is deliberately focused on rather short-term research opportunities drawn directly from our previous experiences, in the scope of this manuscript. Moreover, we depart from the hypothesis that the G∀min∃ SLI is a fundamental building block that will positively contribute to each proposal. I understand the potential amount of work underlying each proposition. However, the creation of a sufficient number of collaborations with PhD students, and other research groups along with a sustained dissemination effort will pave the road towards even more ambitious endeavors built upon some of the bricks discussed in the following paragraphs.

Model-based System Engineering is at the heart of the development process in many companies. However, most of the models used today (even if they are standardized) are often informal. In this direction, in 2014 the INCOSE organization predicted (within 10 years - 2025) the adoption of formal system modeling techniques at all stages of the development cycle (specification, analysis, design, verification) [66]. The integration of formal models mainly promises an increase in confidence in the designed systems while decreasing the costs associated with the production processes (especially in terms of certification). Despite all this will, a set of technical factors is currently slowing down this movement toward the extensive use of the formal. One of these factors is heterogeneity. It appears to be due not only to the intrinsic multidisciplinarity of systems but also, in design, to the abstractions necessary to master complexity. This results in the emergence of many languages covering different aspects of the design space (computation, control, performance, dynamics). This problem of language heterogeneity is not new, it has been studied in the literature at different levels of abstraction and in different application contexts [158, 159, 160, 161]. Even if these efforts bring some answers to the problem of integrating heterogeneous executable models, the use of dedicated (user-defined) formalisms and the need for abstraction in system specifications still pose many difficulties in terms of runtime analysis, notably in terms of migrating analysis tools from one formalism to another.

The theme that I wish to develop is part of the problem of integrating languages and tools, which facilitates both the formalization of languages, the search for efficient analysis algorithms, and the adaptation to industrial constraints. The initial observation that motivates our work is that, in the context of critical system design, the ”software languages” community seems to be divided by two divergent requirements: the formalization of semantics to allow formal analyses and the implementation of efficient execution environments. In the first case, we find software frameworks such as kframework[101] and gemoc studio[102] offering a rich set of tools dedicated to runtime formalization and analysis. However, their runtime performance is limited. Moreover, users are forced to reformulate the semantics through dedicated languages, which increases the

development cost and significantly limits industrial adoption. In the second case, the industry invests heavily in the development of powerful runtime environments that, unfortunately, lack support for formal reasoning. This situation leads to excessive use of model transformation approaches which, in addition to the high cost, increases the risk of semantic deviations between the real execution and that seen by the analysis tools. The model-checking community is probably the most affected by this dichotomy. On the one hand, the industry needs verification tools for "real" languages. On the other hand, the scientific community is pushing for the use of very constrained languages, to simplify the verification problem.

Thus, the main research question that arises is: "Would it be possible to compose in a modular way the existing execution environments (developed by the industry) with generic analysis tools?" Lately, the "model-checking" community seems to be giving a positive answer to this question [103, 104, 1]. However, the problem is more general and covers the set of tools needed to diagnose executable systems throughout their life cycles.

My future work will be oriented toward these aspects. More specifically, I am interested in further **generalizing the G∀min∃ language monitoring** approach in the context of **specification-driven software engineering**. I strongly believe that the next paradigm shift in software engineering is offered by a specification-driven approach, which explicitly separates the conceptual and technological domains.The reification of the semantic interface at the language level, discussed in chapter 2, offers multiple research opportunities, which will contribute to the realization of this vision. In the following we emphasize seven promising directions:

- **Temporal breakpoints**. During the verification phase, temporal logic proved to be the best tool for reasoning about the concurrent and highly non-deterministic specification. With the invention of the multiverse debugging [81], geared toward actor-based formalisms or abstract behavioral specifications [82], the debugger becomes exposed to the difficulties of reasoning about multiple worlds (execution paths) evolving independently and sometimes intersecting, that potentially hide unwanted emerging behaviors. Thus it becomes evident to us that the debugger language should be extended to allow "temporal logic" reasoning. Recently, Maximilian Willembrinck et al. in [162] made the first in this direction introducing time-traveling queries as a solution for collecting user-defined execution data. However, when confronted with multiple execution histories, this approach, which only embodies user-defined collecting semantics, fails to exhibit the time-dependent, causal connections that are offered by temporal logic. In subsection 2.4.2 we have shown a modular architecture for the *Finder* operator, which can easily extend the expressivity of the breakpoint conditions to temporal logic. This extension can be gradual, encoding safety properties as regular expressions over the execution traces, or both safety and liveness properties as either linear temporal logic or buchi automata. While through our approach this extension is trivial, numerous research efforts, in the context of model-checking, show that it can be difficult to correctly write temporal logic specifications [163]. It will be interesting to see if patterns emerge when using temporal logic specifications during debugging. Furthermore, we are forced to observe that currently breakpoint conditions are rather language-specific (even though some similarities exist), offering higher expressivity through temporal breakpoint could benefit from the existence of the IEEE Standard for Property Specification Language (PSL) [164]. For the optimists, the synergy between debug and temporal logic might even offer terrain for cross-fertilization between the software and hardware design worlds.

- **Scheduling-aware functional verification**. Task scheduling is a resource allocation process that is typically studied independently from functional verification. Even when the two processes meet, their joint analysis leads to ad-hoc or rather cumbersome setups, as discussed in section 2.5. Our compositional architecture, presented in section 2.5, has the potential

of improving the scalability of functional verification while reducing the gap between the verification and the execution. In this context, it will be interesting to study the integration between a modular verification tool, such as OBP2, with a scheduling analysis tool, such as Cheddar [165]. Even if, due to the inherent state-space explosion problem, an exhaustive joint analysis might not be scalable, we strongly believe that such a setup can greatly improve the coverage of the functional verification phase through partial explorations of the state-space pruned by the scheduler. From this perspective, the scheduling-aware functional verification approach that we pursue here can be seen as a complementary tool to fuzz testing [166] for identifying security vulnerabilities in mission-critical applications.

- **Fully-verified Language Monitors**. As stated previously, the G∀min∃ formalization is a live formalization effort that crystalizes almost ten years of exploratory research, and which continues to evolve beyond the scope of this document. For instance, recently, in the context of integration of OBP2 verification core in the SDL modeling environment industrialized by PragmaDEV, we have discovered a more elegant way to handle the dependencies between SLI modules, which alleviates the need for the rather cumbersome *collect* function of the interface. The main idea is to provide a truly dependent SLI interface (where required) that is parameterized statically with the *evaluation* function of the subject language and dynamically with subject-language execution steps. The first positive consequence of this novel approach is a cleaner and more robust synchronous composition operator, that allows the dependent SLI to compute internally the available actions. The downside is a slower synchronization operator, which requires language-level caching (on the dependent SLI) to retrieve the previous performances [1]. We are currently investigating the real-world cost of this "more elegant" formalization. Nevertheless, we strongly believe that our formalization finely captures the essence of the subject-language semantics needed for modularly defining practically useful language monitors, as illustrated in section 2.4 and in section 2.5. The next logical challenge, with respect to G∀min∃ SLI, is clearly stating the hypotheses (formalized in the dependent type theory of the lean prover [100]) needed to prove that our operators are sound and that their composition remains sound. The release of lean 4 challenges the boundary between theorem proving and programming [167] and will allow us to step further. We plan to start a project seeking to bridge the gap between our specification and its implementation, an attempt that will result in a fully verified language-agnostic toolkit for executable specification monitoring. Besides the technical proof-engineering challenge [168], a real challenge hides behind the performance penalty incurred by the use of a proof language. Previous attempts to fully formalize an LTL model-checker [169] and a timed automata model-checker [170], using Isabelle/HOL [171] are an order of magnitude slower than their non-formalized counterparts.

- **Hardware model-checkers**. Focus on using hardware platforms dedicated to the implementation of efficient verification algorithms. This axis emerges from the PhD thesis of Emilien Fournier, who has shown impressive results at the algorithmic level [22, 23, 24]. However, further work is necessary to explore the trade-offs related to the implementation of the model. At this level the problem can be seen from 3 perspectives: *(1) The model semantics is mapped in hardware.* In this case, the hardware synthesis time might be prohibitive, if it is not amortized. However, this approach can be beneficial for the verification of circuits, for which only the property and the environment model for the circuit needs to be synthesized. *(2) The model is given by software semantics.* This approach conceptually allows maximizing the reuse, as all existing "software" semantics could be used. However, it seems difficult

[1] NB. For model-checking the evaluation function of the subject language is called multiple times for each state (and there are a lot of states) to decide if the available actions in the property language are enabled.

for a pure software implementation to match the bandwidth of the verification core, which processes one word per cycle. Nevertheless, while the current system-on-chip (SoC) platforms might not be adequate, we believe that a capacity-level analysis should be performed to better characterize the performances needed for a specialized SoC. *(3) The model is given by hardware or hybrid software/hardware semantics.* On this axis, it will be interesting to further investigate the opportunities offered by hardware acceleration for implementing the semantics of certain specification languages, which have features that are hard to compute in software but could benefit from the fine-grained parallelism offered by circuits. This direction might seem daunting, but, our work [24] and [172, 173] show very promising results for the rather simplistic specification language DVE, used by the BEEM model-checking benchmark [174]. Moreover, currently, we are not aware of any systematic study focusing on designing specialized circuits for implementing other executable-specification languages. Besides these semantics considerations, we emphasize that the *hardware model-checker* field is at its infancy and that the promising results (both in terms of states/second and power consumption) shown were obtained on reconfigurable architectures, Xilinx FPGAs to be precise[2]. Thus, we dare to ask, what can be the performance gain on an application-specific integrated circuit (ASIC) solution?

- **Open and dynamic abstraction-refinement environments** should ease the use of abstractions and enable the creation of heterogeneous refinement mappings. Abstraction and refinement are inevitable for managing the complexity, however, their usage in an industrial context remains limited and requires the use of specialized development environments. From our perspective, these tools can be made available to a larger community through the design of language-independent abstractions and the possibility of creating heterogeneous refinement mappings. For instance, zone-based time representation is the core abstraction used to integrate time constraints in specification languages based on timed automata, like Uppaal [175] and Fiacre [88]. Numerous research efforts are focused on transforming a wide range of domain-specific languages into these formalisms for verification [176, 177, 178, 179, 180, 181]. In this context, we strongly believe that capturing the essence of this time abstraction into a reusable semantic component will render this abstraction more accessible to DSL designers. Currently, in the ONEWAY project, we are investigating the design of such a component for allowing real-valued time reasoning in domain-specific business processes targeting the planning phase of product development. Before achieving our goal of a language agnostic operator, the next steps will focus on understanding the dialog between this abstraction and the subject language. Besides the time representation, other widely used abstractions, such as channel [88] and/or clock synchronization [108, 40], could be offered as modules to ease both ES language design and analysis. Moreover, recently, with Matthias Pasquier we have shown that the G∀min∃ SLI allows the language-agnostic implementation of many under-approximations, without impacting the subject language semantics [41]. Furthermore, we are planning to investigate the role that the G∀min∃ *evaluate* function plays for defining heterogeneous refinement mappings between formally-specified digital twins and the underlying running system. As a side-note, the initial motivation for introducing an *evaluate* function over the execution steps came from our work on MoCCML [40], which required reasoning on both the state and the clocks labelling the transitions. State/event based model-checking introduced in [182] formalizes the approach. Temporal logic of actions [183] reifies a similar notion as a step (transition) predicate, named the action. The G∀min∃ *evaluate* captures the essence of these seminal works through the enabled *actions* function and a simple predicate inspired by the *eval* function [184]. Based

[2] https://www.xilinx.com/products/silicon-devices/fpga.html

on this observation, the key insight is that this step evaluation predicate must operate from outside the semantics, from where it can see the steps (source, action, target). It will be interesting to study the impact of a true step-based generalization of the original *eval* [184], forced to look from the outside, on the behavioral reflection in dynamic programming languages.

- Creating **a moldable diagnosis cockpit**. In the literature and during our experimentations we have encountered other highly reusable monitoring components, besides the operators described in section 2.3, that enable further specialization of the language monitors. For instance, the LTSmin [104] introduces *partial-order reduction*, *reordering*, and *caching* as PINS2PINS wrappers (which are equivalent to our language monitors). In several experiments, we have found that a *folding operator* allows the advantageous hiding of parts of the state-space while preserving observability. In this context, the main idea is to define a specialized domain-specific language for designing the diagnosis setup in accordance with the diagnosis engineer's requirements. We can imagine a graphical DSL that based on specifications similar to the ones shown in section 2.5 (see for instance Figure 2.13, and Figure 2.14) will allow the derivation of the required tool. Considering this idea in conjunction with the need for multiple abstraction strategies leads to a monitor-specification language and/or diagnosis environment that would allow the dynamic adaptation of the monitoring setup to suit the diagnostician requirements during the analysis process itself. In this context, the role of the monitor-specification environment could be extended to derive and provide proof of the soundness of the monitoring setup, along with the analysis results.

- **Algebraic specifications of complex algorithms**. Isolating the execution controller from both the language semantics and the monitors brings modularity and reduces the need for temporal reasoning at the operator level. In the context of functional programming languages, similar isolation led to the discovery of *recursion schemes* [185]. More recently, in [186], the authors relied on a similar idea (open recursive style) to propose an elegant architecture for the abstract interpretation of higher-order programming languages. In our context, the reachability execution controller explores the semantics exposed by the SLI, integrating the analysis results in the monitor's state. In his PhD manuscript [93], Emilien Fournier proposed the first generic specification of reachability that subsumes a large class of reachability algorithm implementations, ranging from breadth-first search, and depth-first search to partial and symbolic implementations. One surprising point was that the specification naturally conformed to the G$\forall$min$\exists$ SLI interface, which allowed the use of OBP2 to debug and illustrate the specification behavior. The keys to achieving this result were embracing non-determinism during algorithm design (delaying choice), and a pure dataflow mindset (limit control flow to a minimum) [187]. Preliminary experimentation, following these guidelines, allowed us to obtain a fine-grain model-checker architecture, which further decomposes the traditional buchi model-checking algorithms into a model-checker monitor executed by a simple execution controller. Thus, the model-checker monitor behaves similarly to a scheduler, which linearizes the exhaustive exploration of the semantics. Further investigation will be needed to understand the theoretical and practical implications of these design decisions. Nevertheless, we believe that the algebraic-like specification will greatly simplify the correctness proofs thus providing a strong implementation baseline.

More generally, our theme seeks to identify and apply a systematic approach, guided by the needs of execution analysis for the development and use of system specification languages. At the Lab-STICC laboratory level, this theme is one of the main development axes of the P4S

team. This axis addresses the problem of language heterogeneity through an approach based on the federation of executable formal models. At the national level, our theme is at the confluence of several working groups of the GDR GPL (AFSEC: Formal Approaches to Embedded Communicating Systems, Debugging, IDM: Model Driven Engineering, IE: Requirements Engineering) and the GDR SOC2 (Methods and Tools). Internationally, our theme is clearly identified in the Model Driven Engineering (MODELS) community and the Software Language Engineering (SLE) community. Our unique and ambitious proposal is part of this dynamic, aiming at a wide adoption of formal models. Our future work, on language-language and language-tool integration, seeks to bring composable and sustainable solutions to a key scientific problem for the design of critical embedded systems.

Bibliography

Publications by the Author in International Journals

[1] Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "Unified verification and monitoring of executable UML specifications". In: *Software and Systems Modeling* 20.6 (Dec. 2021), pp. 1825–1855. ISSN: 1619-1374. DOI: 10.1007/s10270-021-00923-9. URL: https://doi.org/10.1007/s10270-021-00923-9.

[2] Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "Modular Scheduling for Verification & Embedded Execution." In: *submitted to Software Testing, Verification and Reliability* (2022).

[3] **Ciprian Teodorov**, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. "Past-Free[ze] reachability analysis: reaching further with DAG-directed exhaustive state-space analysis". In: *Software Testing, Verification and Reliability* 26.7 (2016), pp. 516–542. DOI: https://doi.org/10.1002/stvr.1611. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1611. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1611.

[4] **Ciprian Teodorov**, Philippe Dhaussy, and Luka Le Roux. "Environment-driven reachability for timed systems". In: *International Journal on Software Tools for Technology Transfer* 19.2 (Apr. 2017), pp. 229–245. ISSN: 1433-2787. DOI: 10.1007/s10009-015-0401-2. URL: https://doi.org/10.1007/s10009-015-0401-2.

[5] Lamia Allal, Ghalem Belalem, Philippe Dhaussy, and **Ciprian Teodorov**. "Sequential and Parallel Algorithms for the State Space Exploration". In: *Cybern. Inf. Technol.* 16.1 (Mar. 2016), pp. 3–18. ISSN: 1314-4081. DOI: 10.1515/cait-2016-0001. URL: https://doi.org/10.1515/cait-2016-0001.

[6] Lamia Allal, Ghalem Belalem, Philippe Dhaussy, and **Ciprian Teodorov**. "A Parallel Algorithm for the State Space Exploration". In: *Scalable Computing: Practice and Experience* 17.2 (2016), pp. 129–142. URL: http://www.scpe.org/index.php/scpe/article/view/1161.

[7] Lamia Allal, Ghalem Belalem, Philippe Dhaussy, and **Ciprian Teodorov**. "Distributed algorithm to fight the state explosion problem". In: *International Journal of Internet Technology and Secured Transactions* 8.3 (2018), pp. 398–411. DOI: 10.1504/IJITST.2018.093664. eprint: https://www.inderscienceonline.com/doi/pdf/10.1504/IJITST.2018.093664. URL: https://www.inderscienceonline.com/doi/abs/10.1504/IJITST.2018.093664.

[8] Loïc Lagadec, **Ciprian Teodorov**, and Jean-Christophe Le Lann. "Model-Driven Toolset for Embedded Reconfigurable Cores". In: *Sci. Comput. Program.* 96.P1 (Dec. 2014), pp. 156–174. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.02.015. URL: https://doi.org/10.1016/j.scico.2014.02.015.

[9] **Ciprian Teodorov** and Loïc Lagadec. "Model-driven physical-design automation for FPGAs: fast prototyping and legacy reuse". In: *Software: Practice and Experience* 44.4 (2014), pp. 455–482. DOI: https://doi.org/10.1002/spe.2190. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2190. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2190.

[10] Catherine Dezan, **Ciprian Teodorov**, Loïc Lagadec, Michael Leuchtenburg, Teng Wang, Pritish Narayanan, and Andras Moritz. "Towards a framework for designing applications onto hybrid nano/CMOS fabrics". In: *Microelectronics Journal* 40.4 (2009). European Nano Systems (ENS 2007) International Conference on Superlattices, Nanostructures and Nanodevices (ICSNN 2008), pp. 656–664. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2008.07.072. URL: https://www.sciencedirect.com/science/article/pii/S0026269208004382.

Publications by the Author in International Conferences

[11] Zoé Drey and **Ciprian Teodorov**. "Object-Oriented Design Pattern for DSL Program Monitoring". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 70–83. ISBN: 9781450344470. DOI: 10.1145/2997364.2997373. URL: https://doi.org/10.1145/2997364.2997373.

[12] Tithnara Nicolas Sun, Bastien Drouot, Fahad Rafique Golra, Joël Champeau, Sylvain Guérin, Luka Le Roux, Raúl Mazo, **Ciprian Teodorov**, Lionel Van Aertryck, and Bernard L'Hostis. "A Domain-specific Modeling Framework for Attack Surface Modeling". In: *Proceedings of the 6th International Conference on Information Systems Security and Privacy, ICISSP 2020, Valletta, Malta, February 25-27, 2020*. Ed. by Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp. SCITEPRESS, 2020, pp. 341–348. DOI: 10.5220/0008916203410348. URL: https://doi.org/10.5220/0008916203410348.

[13] Tithnara Nicolas Sun, **Ciprian Teodorov**, and Luka Le Roux. "Operational Design for Advanced Persistent Threats". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450381352. URL: https://doi.org/10.1145/3417990.3420044.

[14] Valentin Besnard, Matthias Brun, Frédéric Jouault, **Ciprian Teodorov**, and Philippe Dhaussy. "Unified LTL Verification and Embedded Execution of UML Models". In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS'18. Copenhagen, Denmark: Association for Computing Machinery, 2018, pp. 112–122. ISBN: 9781450349499. DOI: 10.1145/3239372.3239395. URL: https://doi.org/10.1145/3239372.3239395.

[15] Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "Verifying and Monitoring UML Models with Observer Automata: A Transformation-Free Approach". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS'19)*. 2019, pp. 161–171. DOI: 10.1109/MODELS.2019.000-5.

[16] Frédéric Jouault, Valentin Besnard, Théo Le Calvar, **Ciprian Teodorov**, Matthias Brun, and Jerome Delatour. "Designing, Animating, and Verifying Partial UML Models". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '20. Virtual Event, Canada: Association for Computing Machinery, 2020, pp. 211–217. ISBN: 9781450370196. DOI: 10.1145/3365438.3410967. URL: https://doi.org/10.1145/3365438.3410967.

[17] Frédéric Jouault, Valentin Sebille, Valentin Besnard, Théo Le Calvar, **Ciprian Teodorov**, Matthias Brun, and Jerome Delatour. "AnimUML as a UML Modeling and Verification Teaching Tool". In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2021, pp. 615–619. DOI: 10.1109/MODELS-C53483.2021.00094.

[18] Fahad Rafique Golra, Joël Champeau, and **Ciprian Teodorov**. "Early Validation Framework for Critical and Complex Process-Centric Systems". In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Iris Reinhartz-Berger, Jelena Zdravkovic, Jens Gulden, and Rainer Schmidt. Cham: Springer International Publishing, 2019, pp. 35–50. ISBN: 978-3-030-20618-5.

[19] Mihal Brumbulli, Emmanuel Gaudin, and **Ciprian Teodorov**. "Automatic Verification of BPMN Models". In: *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Toulouse, France, Jan. 2020. URL: https://hal.archives-ouvertes.fr/hal-02441878.

[20] Mihal Brumbulli, Emmanuel Gaudin, and **Ciprian Teodorov**. "Identifying unreachable paths in BPMN models". In: *Complex Systems Design & Management (CSD&M'20)*. Paris, France, Dec. 2020.

[21] Luka Le Roux and **Ciprian Teodorov**. "Partially Bounded Context-Aware Verification". In: *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Vol. 11724. Lecture Notes in Computer Science. Springer, 2019, pp. 532–548. DOI: 10.1007/978-3-030-30446-1_28. URL: https://doi.org/10.1007/978-3-030-30446-1%5C_28.

[22] Emilien Fournier, **Ciprian Teodorov**, and Loïc Lagadec. "Menhir: Generic High-Speed FPGA Model-Checker". In: *2020 23rd Euromicro Conference on Digital System Design (DSD)*. 2020, pp. 65–72. DOI: 10.1109/DSD51259.2020.00022.

[23] Emilien Fournier, **Ciprian Teodorov**, and Loïc Lagadec. "Carnac: Algorithm Variability for Fast Swarm Verification on FPGA". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 2021, pp. 185–189. DOI: 10.1109/FPL53798.2021.00038.

[24] Emilien Fournier, **Ciprian Teodorov**, and Loïc Lagadec. "Dolmen: FPGA Swarm for Safety and Liveness Verification". In: *2022 Design, Automation and Test in Europe Conference (DATE'22)*. 2022.

[25] Valentin Besnard, Frédéric Jouault, Matthias Brun, **Ciprian Teodorov**, Philippe Dhaussy, and Jérôme Delatour. "Modular Deployment of UML Models for V&V Activities and Embedded Execution". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450381352. URL: https://doi.org/10.1145/3417990.3419227.

[26] Valentin Besnard, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. "A Model Checkable UML Soccer Player". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 211–220. DOI: 10.1109/MODELS-C.2019.00035.

[27] Valentin Besnard, Matthias Brun, Frédéric Jouault, **Ciprian Teodorov**, and Philippe Dhaussy. "Embedded UML Model Execution to Bridge the Gap Between Design and Runtime". In: *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Vol. 11176. Lecture Notes in Computer Science. Springer, 2018, pp. 519–528. DOI: 10.1007/978-3-030-04771-9_38. URL: https://doi.org/10.1007/978-3-030-04771-9%5C_38.

[28] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and **Ciprian Teodorov**. "Towards One Model Interpreter for Both Design and Deployment". In: *Proceedings of MODELS 2017 Satellite Event: Workshops (ModComp, ME, EXE, COMMitMDE, MRT, MULTI, GEMOC, MoDeVVa, MDETools, FlexMDE, MDEbug), Posters, Doctoral Symposium, Educator Symposium, ACM Student Research Competition, and Tools and Demonstrations co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA, September, 17, 2017*. Ed. by Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin, and Davide Di Ruscio. Vol. 2019. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 102–108. URL: http://ceur-ws.org/Vol-2019/exe%5C_4.pdf.

[29] Vincent Leildé, Vincent Ribaud, **Ciprian Teodorov**, and Philippe Dhaussy. "A Problem-Oriented Approach to Critical System Design and Diagnosis Support". In: *New Trends in Model and Data Engineering*. Ed. by El Hassan Abdelwahed, Ladjel Bellatreche, Djamal Benslimane, Matteo Golfarelli, Stéphane Jean, Dominique Mery, Kazumi Nakamatsu, and Carlos Ordonez. Cham: Springer International Publishing, 2018, pp. 30–39. ISBN: 978-3-030-02852-7.

[30] Vincent Leildé, Vincent Ribaud, **Ciprian Teodorov**, and Philippe Dhaussy. "Domain-Oriented Verification Management". In: *Model and Data Engineering*. Ed. by El Hassan Abdelwahed, Ladjel Bellatreche, Mattéo Golfarelli, Dominique Méry, and Carlos Ordonez. Cham: Springer International Publishing, 2018, pp. 354–370. ISBN: 978-3-030-00856-7.

[31] Vincent Leildé, Vincent Ribaud, **Ciprian Teodorov**, and Philippe Dhaussy. "A Diagnosis Framework for Critical Systems Verification (Short Paper)". In: *Software Engineering and Formal Methods*. Ed. by Alessandro Cimatti and Marjan Sirjani. Cham: Springer International Publishing, 2017, pp. 394–400. ISBN: 978-3-319-66197-1.

[32] Sebastien Heim, Xavier Dumas, Eric Bonnafous, Philippe Dhaussy, **Ciprian Teodorov**, and Luka Le Roux. "Model Checking of SCADE Designed Systems". In: *8th European Congress on Embedded Real Time Software and Systems (ERTS'16)*. Ed. by ERTS. 2016.

[33] **Ciprian Teodorov**, Luka Leroux, and Philippe Dhaussy. "Context-Aware Verification of a Cruise-Control System". In: *Model and Data Engineering*. Ed. by Yamine Ait Ameur, Ladjel Bellatreche, and George A. Papadopoulos. Cham: Springer International Publishing, 2014, pp. 53–64. ISBN: 978-3-319-11587-0.

[34] Lamia Allal, Ghalem Belalem, Philippe Dhaussy, and **Ciprian Teodorov**. "Using Parallel and Distributed Reachability in Model Checking". In: *Ambient Communications and Computer Systems*. Ed. by Gregorio Martinez Perez, Shailesh Tiwari, Munesh C. Trivedi, and Krishn K. Mishra. Singapore: Springer Singapore, 2018, pp. 143–154. ISBN: 978-981-10-7386-1.

[35] Jean-Philippe Schneider, Joël Champeau, **Ciprian Teodorov**, Eric Senn, and Loïc Lagadec. "A role language to interpret multi-formalism System of systems models". In: *2015 Annual IEEE Systems Conference (SysCon) Proceedings*. 2015, pp. 200–205. DOI: 10.1109/SYSCON.2015.7116752.

[36] Jean-Philippe Schneider, **Ciprian Teodorov**, Eric Senn, and Joël Champeau. "Towards a Dynamic Infrastructure for Playing with Systems of Systems". In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. ECSAW '14. Vienna, Austria: Association for Computing Machinery, 2014. ISBN: 9781450327787. DOI: 10.1145/2642803.2642834. URL: https://doi.org/10.1145/2642803.2642834.

[37] Hiba Hnaini, Luka Le Roux, Joël Champeau, and **Ciprian Teodorov**. "Security property modeling". In: *7th International Conference on Information Systems Security and Privacy (ICISSP 2021)*. Ed. by Steven Furnell Paolo Mori Gabriele Lenzini. 2021.

[38] Khaoula Es-Salhi, Rim S. Boudaoud, **Ciprian Teodorov**, Vincent Ribaud, and Zoé Drey. "KriQL : A Language for Query-based Diagnosis of Transition Systems." In: *15th International Workshop on Automated Verification of Critical Systems (AVoCS'15)*. 2015.

[39] Riwan Cuinat, **Ciprian Teodorov**, and Joel Champeau. "SpecEdit: Projectional Editing for TLA+ Specifications". In: *2020 IEEE Workshop on Formal Requirements (FORMREQ)*. 2020, pp. 1–7. DOI: 10.1109/FORMREQ51202.2020.00008.

[40] Julien Deantoni, Issa Papa Diallo, **Ciprian Teodorov**, Joel Champeau, and Benoit Combemale. "Towards a meta-language for the concurrency concern in DSLs". In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 313–316. DOI: 10.7873/DATE.2015.1052.

[41] Frédéric Jouault, **Ciprian Teodorov**, and Matthias Brun. "Smart Home Model Verification with AnimUML". In: *International workshop on MDE for Smart IoT Systems (MESS'22)*. Nantes, France, 2022.

[42] Théotime Bollengier, Loïc Lagadec, and **Ciprian Teodorov**. "Prototyping FPGA through overlays". In: *2021 IEEE International Workshop on Rapid System Prototyping (RSP)*. 2021, pp. 15–21. DOI: 10.1109/RSP53691.2021.9806222.

[43] Bassirou Diène, Ousmane Diallo, Joel J. P. C. Rodrigues, EL Hadji M. Ndoye, and **Ciprian Teodorov**. "Data Management Mechanisms for IoT: Architecture, Challenges and Solutions". In: *2020 5th International Conference on Smart and Sustainable Technologies (SpliTech)*. 2020, pp. 1–6. DOI: 10.23919/SpliTech49282.2020.9243728.

[44] Ahcene Bounceur, Madani Bezoui, Massinissa Lounis, Reinhardt Euler, and **Ciprian Teodorov**. "A new dominating tree routing algorithm for efficient leader election in IoT networks". In: *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*. 2018, pp. 1–2. DOI: 10.1109/CCNC.2018.8319292.

[45] Erwan Fabiani, Loïc Lagadec, Mohamed Ben Hammouda, and **Ciprian Teodorov**. "Asserting causal properties in High Level Synthesis". In: *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*. 2017, pp. 111–116. DOI: 10.1109/IVSW.2017.8031555.

[46] Siham Rim Boudaoud, Khaoula Es-Salhi, Vincent Ribaudy, and **Ciprian Teodorov**. "Relational and graph queries over a transition system". In: *IEEE EUROCON 2015 - International Conference on Computer as a Tool (EUROCON)*. 2015, pp. 1–6. DOI: 10.1109/EUROCON.2015.7313738.

[47] **Ciprian Teodorov** and Loic Lagadec. "Virtual prototyping of R2D NASIC based FPGA". In: *2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2014, pp. 179–180. DOI: 10.1109/NANOARCH.2014.6880509. URL: https://doi.ieeecomputersociety.org/10.1109/NANOARCH.2014.6880509.

[48] Philippe Dhaussy and **Ciprian Teodorov**. "Context-Aware Verification of a Landing Gear System". In: *ABZ 2014: The Landing Gear Case Study*. Ed. by Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe. Cham: Springer International Publishing, 2014, pp. 52–65. ISBN: 978-3-319-07512-9.

[49] **Ciprian Teodorov** and Loïc Lagadec. "MDE-Based FPGA Physical Design: Fast Model-Driven Prototyping with Smalltalk". In: *Proceedings of the International Workshop on Smalltalk Technologies*. IWST '11. Edinburgh, United Kingdom: Association for Computing Machinery, 2011. ISBN: 9781450310505. DOI: 10.1145/2166929.2166936. URL: https://doi.org/10.1145/2166929.2166936.

[50] **Ciprian Teodorov**, Pritish Narayanan, Loic Lagadec, and Catherine Dezan. "Regular 2D NASIC-based architecture and design space exploration". In: *2011 IEEE/ACM International Symposium on Nanoscale Architectures*. 2011, pp. 70–77. DOI: 10.1109/NANOARCH.2011.5941486.

[51] **Ciprian Teodorov**, Damien Picard, and Loïc Lagadec. "FPGA physical-design automation using Model-Driven Engineering". In: *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. 2011, pp. 1–6. DOI: 10.1109/ReCoSoC.2011.5981495.

[52] **Ciprian Teodorov** and Loïc Lagadec. "FPGA SDK for nanoscale architectures". In: *Proceedings of the 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2011, Montpellier, France, 20-22 June, 2011*. IEEE, 2011, pp. 1–8. DOI: 10.1109/ReCoSoC.2011.5981494. URL: https://doi.org/10.1109/ReCoSoC.2011.5981494.

[53] **Ciprian Teodorov**. "Comparing crossbar-based nano/CMOS architectures". In: *5th International Conference on Design & Technology of Integrated Systems in Nanoscale Era*. 2010, pp. 1–6. DOI: 10.1109/DTIS.2010.5487598.

[54] Damien Picard, Bernard Pottier, and **Ciprian Teodorov**. "Process System Modeling for RSoC". In: *Reconfigurable Communication-centric Systems-on-Chip workshop*. Ed. by J.M. Moreno and G. Sassatelli. Barcelone, Spain, July 2008, Session 6: Mapping and Programming Models. URL: https://hal.archives-ouvertes.fr/hal-00491586.

[55] Cornelia Amariei, **Ciprian Teodorov**, Erwan Fabiani, and Bernard Pottier. "Modeling Sensor Networks as Concurrent Systems". In: *2007 Fourth International Conference on Networked Sensing Systems*. 2007, pp. 296–296. DOI: 10.1109/INSS.2007.4297438.

Publications by the Author in National Conferences

[56] Tithnara Nicolas Sun, Luka Le Roux, **Ciprian Teodorov**, and Philippe Dhaussy. "Exploration de Scénarios de Systèmes Cyber-Physiques pour l'Analyse de la Menace." In: *19e journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'20)*. Ed. by David Delahaye and Ileana Ober. 2020. ISBN: 9782917490303.

[57] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, and **Ciprian Teodorov**. "EMI : Un Interpréteur de Modèles Embarqué pour l'Exécution et la Vérification de Modèles UML." In: *18e journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'20)*. Ed. by David Chemouil and Thomas Lambolais. 2019.

[58] Sebastián Tleye, **Ciprian Teodorov**, Erwan Fabiani, and Loïc Lagadec. "Phadeo : un environnement pour FPGA virtuel". In: *Conférence en Parallélisme, Architecture et Système (COMPAS'15)*. 2015.

References

[59] French National Law. *Arrêté du 23 novembre 1988 relatif à l'habilitation à diriger des recherches*. https://www.legifrance.gouv.fr/loda/id/JORFTEXT000000298904/. 1988.

[60] Aline Dresch, Daniel Pacheco Lacerda, and Jos Antnio Valle Antunes. *Design Science Research: A Method for Science and Technology Advancement*. Springer Publishing Company, Incorporated, 2014. ISBN: 3319073737.

[61] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892.

[62] Object Management Group (OMG). "Semantics of a Foundational Subset for Executable UML Models (FUML)". In: (2021).

[63] Haiyang Zheng. "Operational Semantics of Hybrid Systems". AAI3275674. PhD thesis. USA, 2007. ISBN: 9780549172529.

[64] Leslie Lamport. *Thinking above the code*. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/leslie_lamport.pdf. 2016.

[65] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. "Position paper: the science of deep specification." In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017). DOI: 10.1098/rsta.2016.0331.

[66] International Council on Systems Engineering. *A World In Motion. Systems Engineering Vision - 2025*. Tech. rep. INCOSE, 2014.

[67] Michael J. Pennock and Jon P. Wade. "The Top 10 Illusions of Systems Engineering: A Research Agenda". In: *Procedia Computer Science* 44 (2015). 2015 Conference on Systems Engineering Research, pp. 147–154. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.03.033. URL: https://www.sciencedirect.com/science/article/pii/S1877050915002690.

[68] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[69] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962. ISBN: 0262130114.

[70] Alan C. Kay. "The Early History of Smalltalk". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, pp. 69–95. ISBN: 0897915704. DOI: 10.1145/154766.155364. URL: https://doi.org/10.1145/154766.155364.

[71] Dan Ingalls. "The Lively Kernel: Just for Fun, Let's Take JavaScript Seriously". In: *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS '08. Paphos, Cyprus: Association for Computing Machinery, 2008. ISBN: 9781605582702. DOI: 10.1145/1408681.1408690. URL: https://doi.org/10.1145/1408681.1408690.

[72] Mojtaba Bagherzadeh, Karim Jahed, Benoit Combemale, and Juergen Dingel. "Live Modeling in the Context of State Machine Models and Code Generation". In: *Software and Systems Modeling* (2020), pp. 1–44. DOI: 10.1007/s10270-020-00829-y. URL: https://hal.inria.fr/hal-02942374.

[73] Karen Robson, Kirk Plangger, Jan H. Kietzmann, Ian McCarthy, and Leyland Pitt. "Is it all a game? Understanding the principles of gamification". In: *Business Horizons* 58.4 (2015), pp. 411–420. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2015.03.006.

[74] Amir Kishon, Paul Hudak, and Charles Consel. "Monitoring Semantics: A Formal Framework for Specifying, Implementing, and Reasoning about Execution Monitors". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pp. 338–352. ISBN: 0897914287. DOI: 10.1145/113445.113474. URL: https://doi.org/10.1145/113445.113474.

[75] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201113716.

[76] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. "Omniscient debugging for executable DSLs". In: *Journal of Systems and Software* 137 (2018), pp. 261–288. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.11.025. URL: https://www.sciencedirect.com/science/article/pii/S0164121217302765.

[77] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. "Object-centric debugging". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 485–495. DOI: 10.1109/ICSE.2012.6227167.

[78] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. "Practical domain-specific debuggers using the Moldable Debugger framework". In: *Computer Languages, Systems & Structures* 44, Part A (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 89–113.

[79] Dominik Aumayr, Stefan Marr, Sophie Kaleba, Elisa Gonzalez Boix, and Hanspeter Mössenböck. "Capturing High-level Nondeterminism in Concurrent Programs for Practical Concurrency Model Agnostic Record & Replay". In: *The Art, Science, and Engineering of Programming*. Programming 5.3 (Feb. 2021), p. 39. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2021/5/14.

[80] Jonas Kjær Rask, Frederik Palludan Madsen, Nick Battle, Hugo Daniel Macedo, and Peter Gorm Larsen. "The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions". In: *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021*. Ed. by José Proença and Andrei Paskevich. Vol. 338. EPTCS. 2021, pp. 3–18. DOI: 10.4204/EPTCS.338.3. URL: https://doi.org/10.4204/EPTCS.338.3.

[81] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. "Multiverse Debugging: Non-deterministic Debugging for Non-deterministic Programs". In: *33rd European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Apr. 2019. URL: https://kar.kent.ac.uk/74328/.

[82] Matthias Pasquier, **Ciprian Teodorov**, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. "Practical Multiverse Debugging through User-defined Reductions. Application to UML Models." In: *Proceedings of the 24rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '22. 2022.

[83] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. "Survey and classification of model transformation tools". In: *Software & Systems Modeling* 18.4 (2019), pp. 2361–2397. DOI: 10.1007/s10270-018-0665-6. URL: https://doi.org/10.1007/s10270-018-0665-6.

[84] Javier Troya, Sergio Segura, Lola Burgueño, and Manuel Wimmer. "Model Transformation Testing and Debugging: A Survey". In: *ACM Comput. Surv.* (Feb. 2022). Just Accepted. ISSN: 0360-0300. DOI: 10.1145/3523056. URL: https://doi.org/10.1145/3523056.

[85] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[86] Marc Pierre Joseph Antoni. "Validation d'automatismes ferroviaires de sécurité à base de réseaux de Petri". PhD thesis. Dec. 2009. DOI: 10.24355/dbbs.084-201003051455-0. URL: http://www.digibib.tu-bs.de/?docid=00032597.

[87] Gordon D Plotkin. "The origins of structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60-61 (2004). Structural Operational Semantics, pp. 3–15. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2004.03.009. URL: https://www.sciencedirect.com/science/article/pii/S1567832604000268.

[88] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, M Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. "Fiacre: an Intermediate Language for Model Verification in the Topcased Environment". In: *4th European Congress ERTS Embedded Real Time Software (ERTS 2008)*. Toulouse, France, Jan. 2008, 8p. URL: https://hal.inria.fr/inria-00262442.

[89] Object Management Group. *OMG Business Process Model and Notation (BPMN). Version 2.0.2*. 2013. URL: https://www.omg.org/bpmn/ (visited on 04/27/2021).

[90] PragmaDEV. *PragmaDev Process, a new tool to verify business processes*. online: http://www.pragmadev.com/news/Process_V1.0_EN.pdf. 2019.

[91] PragmaDEV. *A new generation of model checker with PragmaDev Studio V6.0*. online: http://www.pragmadev.com/news/V6.0_En.pdf. 2022.

[92] Frédéric Jouault, Maxime Méré, Matthias Brun, Théo Le Calvar, Matthias Pasquier, and **Ciprian Teodorov**. "From OCL-based Model Static Analysis to Quick Fixes". In: *21st International Workshop on OCL and Textual Modeling (OCL'22)*. Montreal, Canada, Oct. 2022.

[93] Emilien Fournier. "Hardware Acceleration of Safety and Liveness Verification on Reconfigurable Architectures". PhD thesis. ENSTA Bretagne, 2022.

[94] Tithnara Nicolas Sun. "Systems Modeling and Formal Analysis for Advanced Persistent Threats". PhD thesis. ENSTA Bretagne, 2022.

[95] Valentin Besnard. "EMI: Une approche pour unifier l'analyse et l'exécution embarquée à l'aide d'un interpréteur de modèles pilotable". PhD thesis. ENSTA Bretagne, 2020.

[96] Vincent Leilde. "A Diagnosis Support for Formal Verification of Systems". PhD thesis. ENSTA Bretagne, 2019.

[97] Luka Le Roux. "Critical embedded system verification, a non-intrusive approach to divide the initial challenge into a sound set of smaller ones". PhD thesis. ENSTA Bretagne, 2018.

[98] Lamia Allal. "Towards an Efficient Approach for Model-checking with Cloud Computing". PhD thesis. Université Ahmed Ben Bella, Oran 1, 2018.

[99] Jean-Philippe Schneider. "Roles : Dynamic Mediators Between System Models and Simulation Models". PhD thesis. Université de Bretagne occidentale, 2015.

[100] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6.

[101] Grigore Roșu and Traian Florin Șerbănuță. "An overview of the K semantic framework". In: *The Journal of Logic and Algebraic Programming* 79.6 (2010). Membrane computing and programming, pp. 397–434. ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2010.03.012. URL: https://www.sciencedirect.com/science/article/pii/S1567832610000160.

[102] B. Combemale, O. Barais, and A. Wortmann. "Language Engineering with the GEMOC Studio". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 189–191. DOI: 10.1109/ICSAW.2017.61.

[103] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. "Java PathFinder-second generation of a Java model checker". In: *In Proceedings of the Workshop on Advances in Verification*. Citeseer. 2000.

[104] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. "LTSmin: High-Performance Language-Independent Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 692–707. ISBN: 978-3-662-46681-0.

[105] Martín Abadi and Leslie Lamport. "The existence of refinement mappings". In: *Theoretical Computer Science* 82.2 (1991), pp. 253–284. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(91)90224-P. URL: https://www.sciencedirect.com/science/article/pii/030439759190224P.

[106] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 863–868. ISBN: 978-3-642-39799-8.

[107] Leslie Lamport. "Specifying Concurrent Systems with TLA+". In: *Calculational System Design* (Apr. 1999), pp. 183–247. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

[108] André Arnold. "Nivat's processes and their synchronization". In: *Theoretical Computer Science* 281.1 (2002). Selected Papers in honour of Maurice Nivat, pp. 31–36. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(02)00006-3. URL: https://www.sciencedirect.com/science/article/pii/S0304397502000063.

[109] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 336–345. ISBN: 9781450334686. DOI: 10.1145/2737924.2737979. URL: https://doi.org/10.1145/2737924.2737979.

[110] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Representation and Mind Series. The MIT Press, 2008. ISBN: 026202649X.

[111] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. "State/Event-Based Software Model Checking". In: *Integrated Formal Methods*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128–147. ISBN: 978-3-540-24756-2.

[112] Bil Lewis. "Debugging Backwards in Time". In: *Computing Research Repository* cs.SE/0310016 (Oct. 2003).

[113] M. Saxena. "A Language Independent Debugger Semantics Based Debugging in K." MA thesis. University of Illinois at Urbana-Champaign, 2018. URL: https://www.ideals.illinois.edu/handle/2142/101590.

[114] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 675 (1988).

[115] Gérard Berry. "SCADE: Synchronous Design and Validation of Embedded Control Software". In: *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Ed. by S. Ramesh and Prahladavaradan Sampath. Dordrecht: Springer Netherlands, 2007, pp. 19–33. ISBN: 978-1-4020-6254-4. DOI: 10.1007/978-1-4020-6254-4_2.

[116] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN: 193435600X, 9781934356005.

[117] G. J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521.

[118] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. "Model Checking of C and C++ with DIVINE 4". In: *Automated Technology for Verification and Analysis*. Ed. by Deepak D'Souza and K. Narayan Kumar. Cham: Springer International Publishing, 2017, pp. 201–207. ISBN: 978-3-319-68167-2. DOI: 10.1007/978-3-319-68167-2_14.

[119] Van Cam Pham, Ansgar Radermacher, Sébastien Gérard, and Shuai Li. "Complete Code Generation from UML State Machine". In: *MODELSWARD*. 2017. DOI: 10.5220/0006274502080219.

[120] Eran Gery, David Harel, and Eldad Palachi. "Rhapsody: A Complete Life-Cycle Model-Based Development System". In: *Integrated Formal Methods*. Ed. by Michael Butler, Luigia Petre, and Kaisa Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–10. ISBN: 978-3-540-47884-3. DOI: 10.1007/3-540-47884-1_1.

[121] Federico Ciccozzi. "Unicomp: A Semantics-aware Model Compiler for Optimised Predictable Software". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '18. Gothenburg, Sweden: ACM, 2018, pp. 41–44. ISBN: 978-1-4503-5662-6. DOI: 10.1145/3183399.3183406.

[122] Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet. "An Optimized Compilation of UML State Machines". In: *ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Shenzhen, China, Apr. 2012.

[123] Sebastien Revol, Géry Delog, Arnaud Cuccurru, and Jérémie Tatibouët. *Papyrus: Moka Overview*. 2018. URL: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

[124] Tanja Mayerhofer and Philip Langer. "Moliz: A Model Execution Framework for UML Models". In: *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*. MW '12. Innsbruck, Austria: ACM, 2012, 3:1–3:2. ISBN: 978-1-4503-1853-2. DOI: 10.1145/2448076.2448079.

[125] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. "Execution of UML models: a systematic review of research and practice". In: *Software & Systems Modeling* (Apr. 2018). ISSN: 1619-1374. DOI: 10.1007/s10270-018-0675-4.

[126] Petr Ročkai, Zuzana Baranová, Jan Mrázek, Katarína Kejstová, and Jiří Barnat. "Reproducible Execution of POSIX Programs with DiOS". In: *Software Engineering and Formal Methods*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Cham: Springer International Publishing, 2019, pp. 333–349. ISBN: 978-3-030-30446-1. DOI: 10.1007/978-3-030-30446-1_18.

[127] Katarína Kejstová, Petr Ročkai, and Jiří Barnat. "From Model Checking to Runtime Verification and Back". In: *Runtime Verification*. Ed. by Shuvendu Lahiri and Giles Reger. Cham: Springer International Publishing, 2017, pp. 225–240. ISBN: 978-3-319-67531-2. DOI: 10.1007/978-3-319-67531-2_14.

[128] Corina S. Păsăreanu and Neha Rungta. "Symbolic PathFinder: Symbolic Execution of Java Bytecode". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. Antwerp, Belgium: ACM, 2010, pp. 179–180. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859035.

[129] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. June 1989, pp. 353–362. DOI: 10.1109/LICS.1989.39190.

[130] Oksana Tkachuk. "Domain-specific environment generation for modular software model checking". PhD thesis. Kansas State University, 2008.

[131] Christophe Diot, Robert de Simone, and Christian Huitema. "Communication Protocols Development Using ESTEREL". In: *First International HIPPARCH workshop. INRIA Sophia Antipolis* (1994), pp. 15–16.

[132] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". In: *Readings in Hardware/Software Co-Design*. USA: Kluwer Academic Publishers, 2001, pp. 527–543. ISBN: 1558607021. DOI: 10.5555/567003.567050.

[133] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. "Embedded UML Model Execution to Bridge the Gap Between Design and Runtime". In: *MDE@DeRun 2018 : First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems*. Toulouse, France, June 2018.

[134] Andreas Gaiser and Stefan Schwoon. "Comparison of Algorithms for Checking Emptiness on Buechi Automata". In: *CoRR* abs/0910.3766 (2009). arXiv: 0910.3766.

[135] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

[136] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. "A Behavioral Coordination Operator Language (BCOoL)". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ed. by Timothy Lethbridge, Jordi Cabot, and Alexander Egyed. Ottawa, Canada: ACM, Sept. 2015, pp. 186–195.

[137] James B Dabney and Thomas L Harman. *Mastering Simulink*. Pearson, 2004.

[138] Ferenc Belina and Dieter Hogrefe. "The CCITT-specification and description language SDL". In: *Computer Networks and ISDN Systems* 16.4 (1989), pp. 311–341. ISSN: 0169-7552. DOI: 10.1016/0169-7552(89)90078-0.

[139] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596.

[140] Kevin J. Vella and Peter H. Welch. "CSP/occam on Shared Memory Multiprocessor Workstations". In: *22nd World-Occam-and-Transputer-User-Group Technical Meeting (WoTUG-22)*. Ed. by Barry M. Cook. Vol. 57. Concurrent Systems Engineering Series. Amsterdam, the Netherlands: IOS Press, Apr. 1999, pp. 87–119.

[141] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. "Execution Framework of the GEMOC Studio (Tool Demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 84–89. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997384.

[142] Florent Latombe, Xavier Crégut, Benoit Combemale, Julien Deantoni, and Marc Pantel. "Weaving Concurrency in Executable Domain-specific Modeling Languages". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 125–136. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814261.

[143] Benoît Combemale, Julien De Antoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert B. France. "Reifying Concurrency for Executable Metamodeling". In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 365–384. ISBN: 978-3-319-02654-1. DOI: 10.1007/978-3-319-02654-1_20.

[144] Julien DeAntoni and Frédéric Mallet. "TimeSquare: Treat Your Models with Logical Time". In: *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*. TOOLS'12. Prague, Czech Republic: Springer-Verlag, 2012, pp. 34–41. ISBN: 978-3-642-30560-3. DOI: 10.1007/978-3-642-30561-0_4.

[145] Abderraouf Benyahia, Arnaud Cuccuru, Safouan Taha, François Terrier, Frédéric Boulanger, and Sébastien Gérard. "Extending the Standard Execution Model of UML for Real-Time Systems". In: *Distributed, Parallel and Biologically Inspired Systems*. Ed. by Mike Hinchey, Bernd Kleinjohann, Lisa Kleinjohann, Peter A. Lindsay, Franz J. Rammig, Jon Timmis, and Marilyn Wolf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 43–54. ISBN: 978-3-642-15234-4. DOI: 10.1007/978-3-642-15234-4_6.

[146] Richard Barry. *FreeRTOS, Real-time operating system for microcontrollers*. URL: https://www.freertos.org/.

[147] J. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. "Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification". In: *2006 IEEE Conference on Emerging Technologies and Factory Automation*. Sept. 2006, pp. 62–69. DOI: 10.1109/ETFA.2006.355432.

[148] Pavel Parizek and Tomas Kalibera. "Platform-Specific Restrictions on Concurrency in Model Checking of Java Programs". In: *Formal Methods for Industrial Critical Systems*. Ed. by María Alpuente, Byron Cook, and Christophe Joubert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 117–132. ISBN: 978-3-642-04570-7. DOI: 10.1007/978-3-642-04570-7_10.

[149] Thuan Quang Huynh and Abhik Roychoudhury. "A Memory Model Sensitive Checker for C#". In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 476–491. ISBN: 978-3-540-37216-5. DOI: 10.1007/11813040_32.

[150] Arnab De, Abhik Roychoudhury, and Deepak D'Souza. "Java Memory Model Aware Software Validation". In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '08. Atlanta, Georgia: ACM, 2008, pp. 8–14. ISBN: 978-1-60558-382-2. DOI: 10.1145/1512475.1512478.

[151] Robby, Matthew B. Dwyer, and John Hatcliff. "Bogor: An Extensible and Highly-Modular Software Model Checking Framework". In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. Helsinki, Finland: Association for Computing Machinery, 2003, pp. 267–276. ISBN: 1581137435. DOI: 10.1145/940071.940107. URL: https://doi.org/10.1145/940071.940107.

[152] J. Hatcliff, M. B. Dwyer, and Robby. "Domain-specific Model Checking Using The Bogor Framework". In: *Proceedings. 21st IEEE International Conference on Automated Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2006, pp. 369–370. DOI: 10.1109/ASE.2006.34. URL: https://doi.ieeecomputersociety.org/10.1109/ASE.2006.34.

[153] Adam Prout, Joanne M. Atlee, Nancy A. Day, and Pourya Shaker. "Code Generation for a Family of Executable Modelling Notations". In: *Softw. Syst. Model.* 11.2 (May 2012), pp. 251–272. ISSN: 1619-1366. DOI: 10.1007/s10270-010-0176-6. URL: https://doi.org/10.1007/s10270-010-0176-6.

[154] Joanne M. Atlee, Nancy A. Day, Jianwei Niu, Eunsuk Kang, Yun Lu, David Fung, and Leonard Wong. *Metro: An Analysis Toolkit for Template Semantics*. 2006.

[155] Yun Lu, Joanne M. Atlee, Nancy A. Day, and Jianwei Niu. "Mapping Template Semantics to SMV". In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. ASE '04. USA: IEEE Computer Society, 2004, pp. 320–325. ISBN: 0769521312.

[156] Karolina Zurowska and Jürgen Dingel. "A Customizable Execution Engine for Models of Embedded Systems". In: *Revised Selected Papers of the International Workshops on Behavior Modeling – Foundations and Applications - Volume 6368*. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 82–110. ISBN: 9783319219110. DOI: 10.1007/978-3-319-21912-7_4. URL: https://doi.org/10.1007/978-3-319-21912-7_4.

[157] Daniel Balasubramanian, Corina S. Păsăreanu, Gábor Karsai, and Michael R. Lowry. "Polyglot: Systematic Analysis for Multiple Statechart Formalisms". In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'13. Rome, Italy: Springer-Verlag, 2013, pp. 523–529. ISBN: 9783642367410. DOI: 10.1007/978-3-642-36742-7_36. URL: https://doi.org/10.1007/978-3-642-36742-7_36.

[158] Claudius Ptolemaeus, ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

[159] Akos Ledeczi, James Davis, Sandeep Neema, and Aditya Agrawal. "Modeling Methodology for Integrated Simulation of Embedded Systems". In: *ACM Trans. Model. Comput. Simul.* 13.1 (Jan. 2003), pp. 82–103. ISSN: 1049-3301. DOI: 10.1145/778553.778557. URL: https://doi.org/10.1145/778553.778557.

[160] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. "Metropolis: an integrated electronic system design environment". In: *Computer* 36.4 (2003), pp. 45–52. DOI: 10.1109/MC.2003.1193228.

[161] Benoit Combemale, Julien DeAntoni, Benoit Baudry, Robert B. France, Jean-Marc Jézéquel, and Jeff Gray. "Globalizing Modeling Languages". In: *Computer* 47.6 (2014), pp. 68–71. DOI: 10.1109/MC.2014.147.

[162] Maximilian Willembrinck, Steven Costiou, Anne Etien, and Stéphane Ducasse. "Time-Traveling Debugging Queries: Faster Program Exploration". In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 2021, pp. 642–653. DOI: 10.1109/QRS54544.2021.00074.

[163] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Property Specification Patterns for Finite-State Verification". In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*. FMSP '98. Clearwater Beach, Florida, USA: Association for Computing Machinery, 1998, pp. 7–15. ISBN: 0897919548. DOI: 10.1145/298595.298598. URL: https://doi.org/10.1145/298595.298598.

[164] "IEEE Standard for Property Specification Language (PSL)". In: *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010), pp. 1–182. DOI: 10.1109/IEEESTD.2010.5446004.

[165] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. "Cheddar: A Flexible Real Time Scheduling Framework". In: *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*. SIGAda '04. Atlanta, Georgia, USA: Association for Computing Machinery, 2004, pp. 1–8. ISBN: 1581139063. DOI: 10.1145/1032297.1032298. URL: https://doi.org/10.1145/1032297.1032298.

[166] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018), p. 6. DOI: 10.1186/s42400-018-0002-y. URL: https://doi.org/10.1186/s42400-018-0002-y.

[167] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

[168] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. "QED at Large: A Survey of Engineering of Formally Verified Software". In: *Foundations and Trends® in Programming Languages* 5.2-3 (2019), pp. 102–281. ISSN: 2325-1107. DOI: 10.1561/2500000045. URL: http://dx.doi.org/10.1561/2500000045.

[169] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. "A Fully Verified Executable LTL Model Checker". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 463–478. ISBN: 978-3-642-39799-8.

[170] Simon Wimmer and Peter Lammich. "Verified Model Checking of Timed Automata". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 61–78. ISBN: 978-3-319-89960-2.

[171] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540433767.

[172] Mrunal Patel, Shenghsun Cho, Michael Ferdman, and Peter Milder. "Runtime-Programmable Pipelines for Model Checkers on FPGAs". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 51–58. DOI: 10.1109/FPL.2019.00018.

[173] Shenghsun Cho, Mrunal Patel, Michael Ferdman, and Peter Milder. "Practical Model Checking on FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 14.2 (July 2021). ISSN: 1936-7406. DOI: 10.1145/3448272. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/3448272.

[174] Radek Pelánek. "BEEM: Benchmarks for Explicit Model Checkers". In: *Model Checking Software*. Ed. by Dragan Bošnački and Stefan Edelkamp. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 263–267. ISBN: 978-3-540-73370-6.

[175] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "UPPAAL — a tool suite for automatic verification of real-time systems". In: *Hybrid Systems III*. Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243. ISBN: 978-3-540-68334-6.

[176] Doaa Soliman, Kleanthis Thramboulidis, and Georg Frey. "Transformation of Function Block Diagrams to UPPAAL timed automata for the verification of safety applications". In: *Annual Reviews in Control* 36.2 (2012), pp. 338–345. ISSN: 1367-5788. DOI: https://doi.org/10.1016/j.arcontrol.2012.09.015. URL: https://www.sciencedirect.com/science/article/pii/S1367578812000508.

[177] Xiaopu Huang, Qingqing Sun, Jiangwei Li, and Tian Zhang. "MDE-Based Verification of SysML State Machine Diagram by UPPAAL". In: *Trustworthy Computing and Services*. Ed. by Yuyu Yuan, Xu Wu, and Yueming Lu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 490–497. ISBN: 978-3-642-35795-4.

[178] Jagadish Suryadevara, Cristina Seceleanu, Frédéric Mallet, and Paul Pettersson. "Verifying MARTE/CCSL Mode Behaviors Using UPPAAL". In: *Software Engineering and Formal Methods*. Ed. by Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–15. ISBN: 978-3-642-40561-7.

[179] Maruth Ravibanjurdkul, Pittipol Kantavat, and Wiwat Vatanawood. "Transforming YAWL Workflows with Time Constraints into Timed Automata". In: *The 2021 9th International Conference on Computer and Communications Management*. ICCCM '21. Singapore, Singapore: Association for Computing Machinery, 2021, pp. 194–200. ISBN: 9781450390071. DOI: 10.1145/3479162.3479191. URL: https://doi.org/10.1145/3479162.3479191.

[180] Jean-Paul Bodeveix, Mamoun Filali, Manuel Garnacho, Régis Spadotti, and Zhibin Yang. "Towards a verified transformation from AADL to the formal component-based language FIACRE". In: *Science of Computer Programming* 106 (2015). Special Issue: Architecture-Driven Semantic Analysis of Embedded Systems, pp. 30–53. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2015.03.003. URL: https://www.sciencedirect.com/science/article/pii/S0167642315000647.

[181] Subeer Rangra and Emmanuel Gaudin. "SDL to Fiacre translation". In: *Embedded Real Time Software and Systems (ERTS2014)*. 2014, pp. 582–591.

[182] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. "State/Event-Based Software Model Checking". In: *Integrated Formal Methods*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128–147. ISBN: 978-3-540-24756-2.

[183] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: https://doi.org/10.1145/177492.177726.

[184] John McCarthy. "History of LISP". In: *History of programming languages*. 1978, pp. 173–185.

[185] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture*. Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 124–144. ISBN: 978-3-540-47599-6.

[186] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. "Abstracting Definitional Interpreters (Functional Pearl)". In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110256. URL: https://doi.org/10.1145/3110256.

[187] Richard Bird and Jeremy Gibbons. *Algorithm Design with Haskell*. Cambridge University Press, 2020. DOI: 10.1017/9781108869041.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université de Bretagne Occidentale

Titre : G∀min∃ : Exploration de la frontière entre les langages de spécification exécutables et les outils d'analyse du comportement

Mot clés : spécifications, monitoring de langage, analyse comportamentale, model-checking, debugging

Résumé : La communauté de la vérification formelle s'efforce de prouver la conformité d'une spécification par l'utilisation de la logique formelle et de preuves mathématiques. Les progrès considérables réalisés dans les outils de vérification formelle assistée par ordinateur, ainsi que le nombre croissant de réussites, rendent ces méthodes essentielles dans la boîte à outils des concepteurs de systèmes. Cependant, avec l'avènement des modèles et des langages spécifiques à un domaine, un grand nombre de formalismes ont été proposés pour écrire des spécifications de systèmes dynamiques, chacun étant adapté aux besoins spécifiques du domaine ciblé. Une nouvelle question émerge : Comment combler le fossé entre ces formalismes spécifiques au domaine, orientés vers les experts du domaine et les outils de vérification formelle, orientés vers les mathématiciens ? Une des réponses, omniprésente dans la littérature, repose sur l'utilisation de transformations de modèles pour traduire syntaxiquement le modèle spécifique au domaine vers le modèle de vérification. Nous soutenons que cette approche est contre-productive et conduit à une multiplication sémantique, qui nécessite des preuves d'équivalence qui peuvent être difficiles à fournir et à maintenir.

Dans ce manuscrit, je présente une nouvelle réponse au niveau sémantique développée, raffinée et évaluée au cours des 10 dernières années avec l'aide de 6 ingénieurs postdoctoraux, 8 candidats au doctorat et 12 projets collaboratifs. Cette approche, nommée G∀min∃, promet une architecture logicielle modulaire, compositionnelle et réutilisable permettant la conception d'une grande variété d'outils d'exploration du comportement. La brique de base de cette approche est une interface de niveau sémantique agnostique au langage, qui agit comme un pont entre la sémantique dynamique d'un langage spécifique au domaine et les outils d'analyse du comportement. Nous proposons ici une formalisation de l'interface ainsi que quelques opérateurs réutilisables pour la création d'outils d'analyse du comportement pour le débogage interactif, le contrôle de modèle et la surveillance de l'exécution. En plus de passer en revue près d'une décennie de recherches fructueuses, ce document me permet de présenter quelques nouvelles directions de recherche, qui, nous l'espérons, allégeront le poids de la création de nouveaux environnements de conception de spécifications, et rendront le processus de conception plus productifs.

Title: G∀min∃: Exploring the Boundary Between Executable Specification Languages and Behavior Analysis Tools

Keywords: executable specifications, language monitoring, behavioral analysis, model-checking, debugging

Abstract: The formal verification community strives to prove the correctness of a specification using formal logic and mathematical proofs. The tremendous progress in computer-aided formal verification tools, along with an ever-growing number of success stories renders these methods essential in the system designer toolbox. However, with the advent of domain-specific models and languages, many formalisms are proposed for writing dynamic system specifications, each one adapted to the specific needs of the targeted domain. A new question emerges: How to bridge the gap between these domain-specific formalisms, geared toward domain experts, and the formal verification tools, geared towards mathematicians? One of the answers, ubiquitous in the literature, relies on using model transformations to syntactically translate the domain-specific model to the verification model. We argue that this approach is counterproductive leading to semantic multiplication, which requires equivalence proofs that can be hard to provide and maintain.

In this dissertation, I present a new semantic-level answer developed, refined, and evaluated during the last 10 years with the help of 6 postdoctoral fellows, 8 PhD candidates, and 12 collaborative projects. This approach, named G∀min∃, promises a modular, compositional, and reusable software architecture allowing the design of a wide variety of behavior exploration tools. The core building block of this approach is a language agnostic semantic-level interface, which acts as a bridge between the dynamic semantics of a domain-specific language and the behavior analysis tools. Here we propose a formalization of the interface along with some reusable operators for creating behavior analysis tools for interactive debugging, model-checking, and runtime monitoring. Besides reviewing almost a decade of fruitful research, this document allows me to introduce some new research directions, which hopefully will ease the burden of creating novel specification-design environments and render the design process more productive.