TU WIEN Informatics

Behavioral Typing to Support Offline and Online Analysis for Executable DSLs

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Mag. Dorian Leroy
Registration Number 11743117

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Manuel Wimmer
Second advisor: Univ.-Prof. Mag. Dr. Benoit Combemale

External reviewers:
Bernhard Rumpe. RWTH Aachen University, Germany.
Yves Le Traon. University of Luxembourg, Luxembourg.

Vienna, 26th November, 2021

Dorian Leroy	Manuel Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Declaration of Authorship

Mag. Dorian Leroy

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 26th November, 2021

Dorian Leroy

Abstract

Dealing with complexity is an important challenge in software and systems engineering. In particular, designing such systems requires expertise in various heterogeneous domains. Model-Driven Engineering (MDE) is a development paradigm to cope with this complexity through the conception and use of Domain Specific Languages (DSLs). A DSL captures all the concepts required to solve a set of problems belonging to a particular domain. DSLs are geared toward domain experts without requiring experience with programming languages. Using DSLs, domain experts are able to model parts of a system using only concepts of their domain of expertise. A particular category of DSLs, Executable DSLs (xDSLs), go further as they enable, through a provided execution semantics, the definition of dynamic models, which in turn enables early dynamic Verification and Validation (V&V) activities on these models.

All xDSLs share a common need for an ecosystem of tools to create, manipulate, and analyze models. But xDSLs come in many shapes and forms, as each is tailored to a particular domain, both syntactically and semantically. Thus, for each new xDSL, tools must be developed anew, or existing tools adapted. This is a tedious and error-prone task that prompted advances in the field, enabling core and advanced V&V activities for xDSLs in a unifying way through well-defined metaprogramming approaches and generic tools leveraging them. Yet, important aspects of xDSLs and V&V activities stand to benefit from dedicated metaprogramming approaches and generic tooling, respectively. On one hand, no metaprogramming approach currently allows to define the interactions between the models conforming to an xDSL and their environment. On another hand, features at the heart of important V&V activities such as testing and debugging remain challenging to offer in a generic way.

In this thesis, we provide solutions to this problem for a set of tools dedicated to offline and online analysis for xDSLs. This comes under the form of three distinct contributions. First, we provide a new metaprogramming approach to extend the definition of xDSLs to incorporate a clear definition of the possible interactions between conforming models and their environment. Second, we leverage the extended foundations for the definition of xDSLs offered by our metaprogramming approach to provide generic support for offline and online analysis for a broader scope of xDSLs, under the form of trace comprehension operators and runtime monitoring, respectively.

Finally, we leverage the contributions of this thesis to provide an advanced generic modeling environment. We provide implementation details of the various tools derived from our contributions that constitute this modeling environment and illustrate how

they interact with one another in different V&V scenarios. For instance, we show how they can be combined to enable the definition of test scenarios and oracles for models of reactive systems from execution traces collected during an interactive debugging session.

In the context of MDE, where the diversity of xDSLs hampers reuse of tools from one language to another, this thesis aims to extend the foundations upon which generic tools can be defined, and to build upon these extended foundations to provide generic support for defining features of V&V activities such as testing and debugging. This results in an advanced and improved framework in term of V&V for xDSLs, compared to the state of the art.

Contents

CHAPTER 1

Introduction

1.1 Context

The rise of complex, software-intensive systems led to the apparition of new challenges in software and systems engineering, pertaining to their development and maintenance. Due to their complexity, the design of these systems requires stakeholders with diverse and heterogeneous areas of expertise. Each of these disciplines has its own specialized concepts and notations, used by stakeholders to represent as *models* the aspects of a system that are relevant to their field. Managing this heterogeneity requires appropriate methods, methodologies and tools to enable the various stakeholders taking part in a project to communicate and actively participate in the design of the system.

Model-Driven Engineering (MDE) is a development paradigm providing the means to cope with this kind of complexity. One core idea of MDE is going from *descriptive* models representing existing systems to *prescriptive* models that can be used to construct the target system [VBD+13]. On one hand, such models allow to perform early Verification and Validation (V&V) of the designed system, while on the other hand they enable the generation of essential software artifacts. However, this requires to bridge the gap between the problem space (in which stakeholders work) and the solution space (the software artifacts composing the system). To this end, MDE advocates the use of Domain Specific Languages (DSLs), languages of limited expressivity that are built around a set of concepts, notations, and tools suitable for modeling the aspects of a system related to their respective domain.

The use of DSLs allows modelers (*i.e.,* domain experts, stakeholders) to take an active part in the design of systems, through the provision of models. From these models, the application of automated techniques such as code generation enables the direct production of essential parts of the system. While many DSLs are used to model the structural aspects of a system, a large amount of so-called Executable DSLs (xDSLs) are dedicated to modeling their behavioral aspects, under the form of *behavioral models* (*e.g.,* [Obj13b, BCCG07, FNTZ00, HLN+90, OAS07]).

Over the course of the execution, the structure of a behavioral model translates into behavior that alters parts of the model and exchanges messages with its environment. Understanding the *meaning* of such models cannot be done by inspecting their structure, as is the case for structural models. This means that, to ensure the correctness of these models with regards to their expected behavior, techniques for behavioral analysis are necessary, such as debugging [BLC+18], runtime verification [LS09] and testing [DK07]. Therefore, to be properly usable, xDSLs require additional tool support with regards to their non-executable counterparts, which incurs extra development and maintenance costs.

The Software Language Engineering (SLE) research field emerged precisely to provide tools and approaches that deal with such obstacles to the adoption of DSLs in software and systems engineering. It is defined by Kleppe as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [Kle08]. A recent development in SLE are language protocols (*e.g.,* Debug Adapter Protocol, Language Server Protocol, etc.), which govern the edition, execution and debugging of models, and facilitate both the provision of tools related to these modeling activities, and their integration within existing Integrated Development Environments (IDEs) such as Eclipse and VSCode.

1.2 Problem Statement

But xDSLs come in many shapes and forms: from graphical to textual, imperative to declarative, data flow to control flow, they encompass every kind of software languages. This diversity hampers the reuse of tools from one xDSLs to another, which in turn often imposes to develop such tools anew for new xDSLs. This is a tedious, error-prone and repetitive task, on which little manpower can usually be spent.

Thus, one way to incentivize the adoption of xDSLs is to provide cost-free tool support with state-of-the-art capabilities, applicable out of the box to any xDSL, while still allowing to manipulate domain concepts. To this end, dedicated concepts and tools have emerged as part of an ongoing community effort. Such advances include model animation, trace recording and querying, and omniscient debugging, all defined in a generic or generative way to be readily applicable to a wide range of xDSLs. Yet, several challenges still need to be tackled to obtain widely applicable behavioral analysis facilities for xDSLs that are on par with those available for General Purpose Languages (GPLs). In this thesis, we identify two such challenges, presented thereafter, and focus on addressing them.

The first challenge we identify consists in extending the scope of the xDSLs supported by state-of-the-art generic tools and workbenches to also include so-called *reactive DSLs*. These are xDSLs whose models can interact with their environment during their execution. However, this aspect of executable models, when it is not ignored altogether, is often supported in an *ad hoc* way. This means that interaction-centric tools are designed for specific DSLs and can hardly be reused for other reactive DSLs. If realized in a *unified* way instead, interaction-centric tools could be reused across a wide range of xDSLs. But

the heterogeneity of how the execution semantics of xDSLs is defined makes the provision of such a unified approach challenging.

The second challenge stems from the fact that a number of core online and offline behavioral analysis facilities available to most GPLs have yet to be proposed as part of the aforementioned community effort. On one hand, offline facilities allow modelers to investigate the behavior of a model under a specific execution scenario from its execution trace. Such facilities would for instance allow to look for potential cycles and bottlenecks, or to evaluate the impacts of a given model change by comparing execution traces. On the other hand, online facilities allow both manual and automated analysis of the behavior of a model while it is running. Examples include conditional breakpoints, test cases and test suites for executable models. Again, the heterogeneity of xDSLs makes the provision of such facilities in a generic way challenging.

We summarize these two orthogonal challenges as follows:

Challenge#1 Including reactive DSLs in the scope of xDSLs supported by existing and future generic tools for behavioral analysis requires a unified way to both define and implement the interactions that conforming models have with their environment.

Challenge#2 The provision of competitive IDEs for xDSLs necessitates online and offline behavioral analysis facilities that work at the domain level and yet are applicable to a wide range of xDSLs.

1.3 Behavioral Analysis Facilities for Reactive DSLs

To address these two challenges, we organize our approach around three contributions. First, we investigate a novel language engineering approach to extend the specification of xDSLs with the definition of their *behavioral type*. Behavioral types of xDSLs expose the possible interactions with their respective conforming models. At the core of the approach is a metalanguage that is used to define behavioral types in a unified way across all xDSLs. Implementation relationships can then be defined between these behavioral types and various xDSLs, the same type being implementable by different DSLs. In addition, we provide the possibility to define type hierarchies through subtyping relationships between behavioral type. The operational semantics of our metalanguage comes under the form of an event manager configured by the implementation and subtyping relationships. At runtime, this event manager translates the received interactions into actual behavior in terms of the execution semantics of the DSL to which running models conform. This generic way to define and expose the behavioral type of xDSLs in turns allows to define generic, interaction-centric tools.

Second, we define a set of operators to manipulate execution traces recorded during the execution of models conforming to xDSLs. These formally defined operators facilitate the analysis of execution traces, helping filtering noisy data and folding successive equivalent states. They also allow trace comparison, as well as cycle and bottleneck detection through an operator building the state graph corresponding to a given execution trace.

These operators can be used to compare how an updated version of a model fares with regard to older versions, and to identify execution states of interest.

Third, we propose a backend for the monitoring of temporal properties defined over executable models. This backend integrates structural queries on the dynamic state of running models with the detection of temporal patterns through Complex Event Processing (CEP). The structural queries leverage the definition of DSLs to allow modelers to use domain concepts when defining their properties. On top of this integration, we provide a temporal property language relying on the widely used Property Specification Patterns (PSPs) to define the temporal aspect of properties. Properties defined with the provided language are translated into CEP statements. The proposed backend offers facilities to register new properties to monitor, and to receive notifications upon the validation or violation of monitored properties during the execution. These facilities enable the development of a wide array of generic tools observing the execution and waiting for specific conditions to be met to perform some action (*e.g.,* pausing or stopping the execution, generating a report, sending an event, etc.).

1.4 Ecosystem of Behavioral Analysis Facilities for Reactive DSLs

We leverage the contributions of this thesis to provide a generic modeling environment supported by an ecosystem of tools for behavioral analysis.

First, we provide an event injection GUI. This component provides an event occurrence configurator based on the definition of the behavioral types of the xDSLs of running models. This event occurrence configurator allows to configure domain-specific stimuli through a dynamically generated GUI, and to send them to running models. This component also lists all the stimuli emitted by running models. It pairs well with debuggers, as being able to suspend the execution of a model under specific circumstances allows modelers to evaluate the impact of sending event occurrences to it.

Second, we provide a fluent Application Programming Interface (API) intended for a programmatic use of the trace comprehension operators. Atop this API, we build a set of visualizations for their possible outputs. This set of visualizations allow manipulation and comparison of execution traces, as well as building of a graphical representation of the state graph corresponding to an execution trace.

Third, we provide a property monitoring GUI. This component allows to load temporal properties defined with our temporal property language, and observe whether they are satisfied or violated by a particular model. This component also allows temporal properties to be used as temporal conditional breakpoints during debugging sessions.

Finally, by combining behavioral types, runtime monitoring, and the trace manipulation and analysis operators, we provide support for the definition of test cases for executable models. Such test cases consist of an interleaving of stimuli to be sent to the running model on one hand, and of checkpoints on the other hand. Such checkpoints specify the stimuli expected to be received from the model, the temporal properties

that must be satisfied, or a trace comparison. The list of the checkpoints of a test case constitutes its oracle.

1.5 Context of this Thesis

This thesis took place as part of an international PhD agreement between TU Wien and University of Rennes 1. On the austrian side, it is partially funded by the Austrian Science Fund (FWF): P 28519-N31, as well as by the Austrian Agency for International Mobility and Cooperation in Education, Science and Research (OeAD) on behalf of the Federal Ministry for Science, Research and Economy (BMWFW) under the grant number FR 08/2017. On the french side, it is partially funded by the French Ministries of Foreign Affairs and International Development (MAEDI) and the French Ministry of Education, Higher Education and Research (MESRI).

1.6 Outline

Chapter 2 introduces the foundations of software language engineering, model-driven engineering, executable metamodeling, runtime monitoring and complex event processing.

Chapter 3 provides an overview of the state of the art of dynamic verification and validation in executable metamodeling, and the proposal that led to our contributions.

Chapter 4 presents the foundational contribution of this thesis, whose purpose is to extend the definition of executable domain-specific languages to incorporate the stimuli their models can exchange with their environment.

Chapter 5 presents our contributions on trace comprehension, which consists of four operators to manipulate, analyze and compare execution traces obtained from models conforming to executable domain-specific languages.

Chapter 6 presents our contribution on runtime monitoring for executable domain-specific languages, which provide a generic backend for runtime monitoring and a temporal property language built on top of this backend to facilitate the definition of such properties.

Chapter 7 presents how we leveraged our three contributions to provide tools for advanced verification and validation for executable domain-specific languages, including an event management facility, temporal property monitoring and execution trace visualization facilities. In this chapter, we also provide a vision of how a better integration of these tools can be achieved to provide an enhanced modeling workbench.

Chapter 8 concludes the thesis by summarizing the advances it brings to the field of software language engineering and verification and validation for executable

domain-specific languages. We end by discussing several perspectives of future research on the topic.

CHAPTER 2

Background

In this chapter, we introduce the foundations upon which our contributions and their applications are built. In Section 2.1, we first introduce the main tools used to engineer software: software languages. We then cover the field of Software Language Engineering (SLE) in Section 2.2, which consist in defining new approaches pertaining to the various activities involved in the life-cycle of software languages, and more precisely of Domain Specific Languages (DSLs). Next, we introduce Model-Driven Engineering (MDE) in Section 2.3, which advocates for the separation of concerns through the use of multiple DSLs, and aims to bridge the semantic gap between the problem and the solution domain through techniques such as code generation. In Section 2.4, we further dig into MDE by presenting the field of executable metamodeling, which enables the analysis of behavioral models by adjoining an execution semantics to the specification of DSLs used to describe the behavior of systems. Finally, in Section 2.5, we cover the most widely used techniques for the behavioral analysis of software systems.

2.1 Software Languages

Software languages are at the heart of numerous fields of computer science and software engineering. The term *software language* includes all the languages used in the development of software systems: from programming languages, to modeling languages, to configuration languages and formal languages [Kle08]. Independently to these various kinds of languages, we distinguish two main categories of software languages: General Purpose Languages (GPLs) and Domain Specific Languages (DSLs).

On one hand, GPLs provide a vast degree of expressiveness and can be used to handle a wide panel of concerns in many application domains. However, their great expressiveness is limited to the solution domain, and they lack specialized notations and concepts tailored to specific domains of application (*i.e.,* problem domains). This means that the ones that are able to get the most out of GPLs are software experts, who seldom

are experts of the problem domain as well. For this reason, while GPLs can be used to develop any kind of software system, they incur additional difficulties when software engineers must correctly map the problem domain to the solution domain. In particular, the lack of abstraction over the problem domain hinders the communication between software engineers and domain experts.

On the other hand, DSLs generally have very limited expressiveness, but are dedicated to a particular field of expertise —their domain— and are thus tailored for experts of that domain [MHS05, Fow10]. As such, their expressiveness is restricted to the concepts pertaining to their application domain. While restricting the problems they can solve to a particular class of problems, this specialized expressiveness makes models defined with a DSL more concise, intuitive and easier to understand and maintain [DK98]. In addition, as such models refer to concepts from the problem domain, domain experts can take a more active part in the development process [VDKV00]. The domain of application of DSLs varies widely, from very restricted (*e.g.,* the HTML syntax is dedicated to design documents displayed in a web browser) to widely applicable (*e.g.,* DSLs for statechart modeling can be used in many domains such as internet of things, user interfaces and parsing). DSLs have been and keep being developed, maintained and used in numerous domains [dNVN+12], and are the cornerstone of language-oriented programming [War94, Fow05] and language-oriented modeling [Com15]. To enable and facilitate its use, a DSL requires a dedicated ecosystem of tools and services supporting users that write, analyze and manipulate models defined with the language. These tools, as do their GPL counterparts, support the development process by facilitating model comprehension, and by automating tedious tasks. However, supplying such an ecosystem of tools for a DSL, either from scratch or by adapting existing tools to the specificities of the DSL, is an expensive and error-prone task [Voe14].

When language engineers build a new DSL, they generally choose one of two options regarding the incarnation of the new language: they build either an *external* or *internal* DSL. Assembling an external DSL involves the creation of a new language and its ecosystem of inter-operating tools: editors, compilers, interpreters, analyzers, etc. Syntax and semantics are specified by language engineers using languages dedicated to the specification of new languages [VWT+14], usually known as metalanguages.

The creation of internal DSLs differs in that the new DSL defined as part of an existing language, known as the *host language* [H+96]. The high-level domain concepts of the new DSL are encoded using the language constructs of the host language, which in turn allows gives access to the infrastructure it provides. Editors, compilers, or interpreters of the host language can be reused as is, which significantly lowers the development costs compared to external DSLs. But this means that internal DSLs cannot do more than their host language: they have to contend with the programming paradigm, type system and tooling provided by their host language. Selecting an appropriate host language is thus of capital importance [RG09]. The concrete form of an internal DSL may be a fluent Application Programming Interface (API), or rely on implementation techniques introducing new syntactic constructs over a GPL (e.g., using macros [BS02], desugaring [ERKO11], extensible compilers [NCM03], or meta-objets [TCKI99]).

Other authors argue that language designers should not be forced into choosing a particular shape once and for all, but that the shape of a given DSL should adapt according to particular users and uses [CTI15]. Such DSLs are named metamorphic DSLs [ACC14]. In this thesis, we focus on external DSLs, and refer to them using the term DSL.

2.2 Software Language Engineering

Developing new software languages is known to be a very complex task that can last for as long as the language is in use, with potentially several updates released each year (*e.g.,* Java). A number of techniques dedicated to the development of DSLs exist to ease their engineering process, taking into account the specificities of these languages [Spi01, VBD+13]. As "software languages are software too" [FGLP10], the development of DSLs also inherits from the complexity of software development in general. Concerns such as language maintainability, reusability or evolution all have an impact on their engineering process. In addition, the usability of a DSL is of crucial importance to its users. Thus, the development of DSLs also includes the provision of Integrated Development Environments (IDEs) and services aiming to maximize the productivity of language users.

As for any tool, the benefits of using DSLs must outweigh their engineering costs, but their reduced scope of application further reinforces the need to lower such costs. The use of time-proven software engineering techniques to facilitate the development of DSLs is thus most fitting. This led to the emergence of the SLE discipline, defined by Kleppe as "the application of systematic, disciplined, and measurable approaches to the development, use, deployment, and maintenance of software languages" [Kle08]. Much like GPLs, DSLs can be broken down into three distinct parts: their abstract syntax, their concrete syntax, and their semantics. [HR04]. In this section, we review how SLE supports the definition of these artifacts.

2.2.1 Abstract Syntax

The abstract syntax is the core element of a language, as it defines its syntactic constructs independently from their potentially protean representation [McC93]. Based on the abstract syntax of a language, tools such as parsers and compilers are able to build an internal representation of a program or model expressed with that language. This internal representation is called an Abstract Syntax Tree (AST) and can processed by other tools such as analyzers. Essentially, the abstract syntax of a language defines the set of all the ASTs that can be expressed with that language.

Specifying the abstract syntax of a software language is usually done using a dedicated metalanguage. One of two formalisms are generally used for the specification of abstract syntaxes: grammars and metamodels. We hereby focus on the use of grammars and defer the presentation of metamodels to Section 2.3. The most widespread way of specifying the context-free grammars of software languages to this day remains the Backus-Naur

Form (BNF) and Extended BNF notations, originally introduced to describe the syntax of ALGOL in the ALGOL report [BBG+60]. The BNF notation specifies both the abstract syntax of a language and its textual representation (*i.e.,* its keywords), while McCarthy later made a case for the separation of abstract and concrete syntax [McC93].

Grammars also play an important role from a software language engineering perspective, as they serve as the primary input for programs generating compilers from language specifications, *i.e.,* compiler-compilers. For instance, parsers generators (e.g., Yacc [J+75], ANTLR [Par13]) are programs that can automatically generate a language-specific parser from a grammar specification. Spoofax [KV10] and MontiCore [KRV10] are two language workbenches that derive a set of classes corresponding to the provided grammar. After parsing, the resulting ASTs consist of sets of instances of those classes. Many language workbenches also automatically derive editor support (including *e.g.,* syntax highlighting and document outline) from a grammar specification. Others, such as Xtext [EB10], integrate grammars and metamodels through a unified formalism. In this thesis, we focus on DSLs whose abstract syntax is defined through metamodels, which are further developed in Section 2.3.

2.2.2 Concrete Syntax

Where the abstract syntax of a language defines its concepts, the concrete syntax defines how these concepts are represented an manipulated by language users. Concrete syntax may either be textual or graphical. In the former case, programs are represented as a sequence of characters, whereas in the latter case, programs are represented as a graphical layout of arbitrary symbols (*e.g.,* BPEL [JEA+07], Simulink [DH04]). It is not unusual for a language to have both a textual and a graphical syntax defined, or even several of each. Since the contributions and applications of this thesis are agnostic to the concrete syntax of languages, we only briefly present textual and graphical syntaxes.

As explained previously in Section 2.2.1, metalanguages allowing to define abstract syntax as a grammar (*i.e.,* those based on the (E)BNF formalism) also define a textual concrete syntax for this language. Terminal symbols of a grammar define the concrete keywords which users use to build syntactically correct programs, following the grammar rules.

Numerous modeling languages are manipulated through a graphical concrete syntax. The UML specifications, for instance, directly specify the notations that must be employed for each UML diagram [OMG13]. In this case, model editing consists in the manipulation of graphical shapes (e.g., boxes, arrows, actors). The Sirius framework [VMP14] provides a metalanguage to directly define graphical concrete syntaxes for languages, in turn allowing to derive a graphical editor for their conforming models and programs.

2.2.3 Semantics

In SLE, semantics is used to give meaning to the models one can define from the abstract syntax of a language. Formal semantics specifications can be leveraged to provide tooling for languages, such as parsers, compilers or type checkers, in either a generic or a

generative way. The semantics of a software language is composed of the *static* semantics of the language and of its *dynamic* semantics.

On one hand, the static semantics of a language specifies a set of structural constraints on programs that cannot usually be expressed with the metalanguages used to defined the abstract syntax of the language. Such constraints define the valid subset of all the ASTs that can be defined for the abstract syntax of a given language. These constraints can include checking the uniqueness of all variable declaration within a scope, checking that every identifier is declared before it is used, and so on. More complex constraints such as type systems are also often part of the static semantics of languages.

On the other hand, the dynamic semantics of a language specifies the runtime behavior of its programs. While there are three prominent approaches to define such dynamic semantics [Mos01], these approaches are not mutually exclusive. On the contrary, the specificities of each approach make them suitable for different purposes. We hereby provide a brief overview of these approaches.

Axiomatic Semantics *Axiomatic semantics* consist of a set of predicates over the abstract syntax of the language, called axioms [Flo93]. One way to define an axiomatic semantics is to use Hoare triples [Hoa69]. Axiomatic semantics can be considered as a specification of the logical properties of a program rather than of its precise meaning. This makes axiomatic semantics particularly suitable for the use of formal methods.

Translational Semantics *Translational semantics* consist of an exogenous transformation from the abstract syntax of the source language to the abstract syntax of a target language whose dynamic semantics if well-defined. Compilers are the most widespread example of translational semantics. When the translational semantics constructs mathematical objects (called denotations), the term denotational semantics is usually preferred [Sch87].

Operational Semantics *Operational semantics* define the meaning of programs in terms of how their execution unfolds, as a sequence of computational steps. It comes in two styles: the *big-step* style, also known as *natural semantics*, and the *small-step* style. Natural semantics directly relates a syntactic construct (*e.g.,* an expression, an instruction) to the result of its execution. Conversely, small-step operational semantics detail what the next execution state will be from the current one, thereby specifying the complete sequence of computational steps taking place during the execution of a program.

2.2.4 Language Workbenches

The engineering of new software languages requires a multitude of metalanguages each tailored to address specific implementation concerns of a DSL, and tools such as generators, analyzers, etc. All these are commonly brought together in a unified environment called a language workbench, a term originating from the work of Martin Fowler [Fow05]. The

aim of language workbenches is to provide all the services required for both developing new software languages, and using these languages.

Numerous language workbenches have been developed for various technological spaces: Rascal [KVDSV09], GME [LMB+01], Monticore [KRV10, GKR+08, KRV08], Spoofax [KV10, VWT+14, WKV14], LISA [Mer13], Neverlang [VC15], ASF+SDF [vdBvDH+01], MPS, etc. In [EVDSV+15], Erdweg *et al.* perform a study of the features offered by different popular language workbenches. This study showed that while most features related to the design of DSLs and code generation from models are overall well supported, features related to the behavioral analysis of conforming models are less widely supported.

2.3 Model-Driven Engineering

MDE is a development paradigm whose purpose is to facilitate the development of complex software systems [Sch06, FR07]. Complex systems generally involve numerous stakeholders from a great diversity of domains (*e.g.,* security, physics), each contributing to specific aspects of the system. Integrating the domain knowledge of all the involved stakeholders (or *domain experts*) in a single system is a challenging engineering task. Indeed, this requires to bridge the gap between the high-level concepts used by domain experts, and the low-level concepts provided by the programming languages used to implement the system.

To overcome this gap, MDE promotes the use of multiple DSLs, each providing abstractions and tools that allow domain experts to use domain concepts they are familiar with to model the corresponding aspects of the system [MHS05]. Automated techniques can then be used to generate concrete software artifacts from the domain models of the system.

The key idea of MDE is thus to go from the descriptive models that domain experts are used to, to prescriptive models that can be used to build the final system [Béz04, Béz05]. This in turn enables domain experts to perform early Verification and Validation (V&V)— through techniques such as simulation or model checking [CJGK+18]— on such models directly. Domain experts are thus able to ensure the correctness of their models and root out their defects at the domain level, before the corresponding software artifacts are integrated into the complete system.

Because the application of MDE requires to design and use an array of DSLs, a number of techniques have been developed for their engineering. In the remainder of this section, we introduce both the fundamental concepts of MDE and the aforementioned SLE techniques used in MDE.

2.3.1 Metamodel

At the core of MDE is the *metamodeling* activity, which consists in defining *metamodels* [AK03]. In the MDE paradigm, metamodels are the centerpiece of DSLs, as they are used to define their abstract syntax [CSW08]. Metamodels precisely define the concepts of a specific domain and the relationships between them. In object-oriented metamodels,

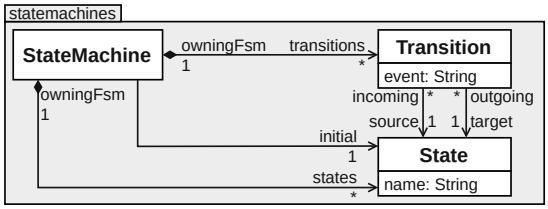


Figure 2.1: Example metamodel for state machines.

these domain concepts are reified as classes usually designated as *metaclasses*. These metaclasses contain properties that can either be *attributes* holding primitive or enumerated values, or *references* toward other metaclasses. Metaclasses can *inherit* from one another, as is possible in object-oriented programming languages. Metaclasses can also be defined as *abstract*, to signify that they cannot be directly instantiated as part of models and must be specialized through inheritance.

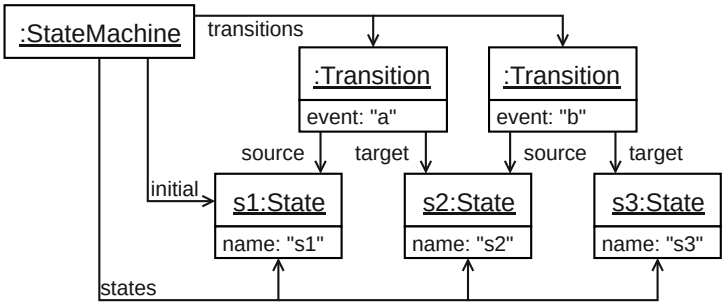
Figure 2.1 shows the metamodel defining the abstract syntax of a State Machine language, using a class diagram notation. This metamodel contains 3 metaclasses corresponding to the concepts of the language: state machines (StateMachine), states (State), and transitions (Transition). A state machine is composed of a set of states, one of which is the initial state, and of a set of transitions between these states. States are identified by their name and transitions are fired when they receive a particular event.

To some extent, most metamodeling languages (*i.e.,* languages designed to define metamodels) allow to define parts of the static semantics of a language, for instance through multiplicities and containment references. Yet, metamodeling language do not always provide the necessary expressivity to define the complete static semantics of a language. When that is the case, additional languages such as OCL [Obj14] can be used to specify the remaining constraints.

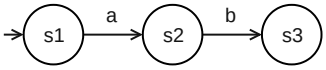
In the modeling community, the Essential Meta-Object Facility (EMOF) [Obj16] is a widely used standard for object-oriented metamodeling, originating from the Object Management Group (OMG). EMOF is supported by the Object Constraint Language (OCL) [Obj14], which enables the definition of complex static semantics rule and is also maintained by the OMG. The Ecore language from the Eclipse Modeling Framework (EMF) [SBMP08], which benefits from extensive tooling, is very closely aligned with EMOF. This makes it the *de facto* standard for metamodel definition: many popular tools such as ATL [JABK08], Xtext [EB10], Kermeta [JCB+13], and Epsilon [KDRPP09] are built on top of —or interoperable with— EMF and Ecore.

2.3.2 Model

With metamodels defined, a definition can be given to models in the MDE sense. A model can be defined as a set of objects (called *model elements*) that are each an instance of a metaclass defined in a metamodel. We say that a given model *conforms to* a given metamodel if each of its elements are valid instances of a concrete metaclass defined in the metamodel, and if all the relations between these model elements conform to the



(a) Represented as an object diagram.



(b) Represented with a graphical concrete syntax.

Figure 2.2: Example of state machine model.

structural constraints constituting the static semantics of the metamodel. As metamodels are an essential component of DSLs, we also say that a model conforms to a DSL, which is a shorter way of saying that a model conforms to the metamodel of a DSL.

Figure 2.2 shows an example of model conforming to the State Machine metamodel shown in Figure 2.1. Figure 2.2a represents this model as an object diagram that shows all the instances (*i.e.,* the model elements) it contains. The model contains 3 instances of State, 2 instances of Transition and a single instance of StateMachine. Figure 2.2b shows a representation of this same model using a graphical concrete syntax which draws states as labeled circles and transitions as labeled arrows. In addition, the initial state is identified by an incoming arrow.

2.3.3 Model Transformations

Model transformations are key to the success of MDE, to the point that they have been qualified as its "heart and soul" [SK03]. Model transformations are programs dedicated to the manipulation of models. As such, they enable the automation of recurring modeling activities such as model refactoring [ZLG05], slicing [BCBB15], code generation and many more [CH03, MVG06]. As a result, they have been extensively studied as first class artifacts [CH06, DREP12, KSB08].

Model transformations are executed on an arbitrary number of input models and produce an arbitrary number of output models (if any). The input and output models of a model transformation must conform to its corresponding input and output metamodels. Model transformations can be defined using any programming paradigms, such as declarative programming (*e.g.,* QVT-R [Omg08], VIATRA [CHM+02]), imperative programming

(*e.g.,* Xtend/EMF, Kermeta [JCB+13]), triple graph grammars (*e.g.,* [Sch94]), or hybrid (ETL [KPP08], ATL [JABK08]). Model transformations operate through *transformation rules*, each defining a set of changes to be performed on a subset of the model elements of input models.

Model transformations come in multiple kinds. When both the input and output models conform to the same metamodel, model transformations are said to be *endogenous*. Conversely, when the input and output metamodels differ, model transformations are *exogenous*. Finally, if a model transformation directly changes the input model without creating an output model, it is said to be an *in-place* transformation. By definition, in-place transformations are always endogenous.

In this thesis, we use model transformations to define the operational and translational semantics of languages.

2.4 Executable Metamodeling

While in MDE, many DSLs are used to define models representing the structural aspects of systems, a large amount are used to express their behavioral aspects. To fully adhere to the philosophy of MDE, domain experts using such DSLs should be able to use tools to explore the runtime behavior of their models at the domain level. For this, executable metamodeling advocates the engineering and use of *Executable DSLs (xDSLs)* to define *executable models*.

Engineering an xDSL requires to provide an execution (or dynamic) semantics for it at the domain level. In return, xDSLs offer two main benefits. First, they enable the use of *dynamic* analysis tools at the domain level, to ensure that an executable model is correct with regard to its intended behavior. This allows domain experts to simulate, animate, test and debug their models[1] using the concepts of their field of expertise. Second, model executability gives the possibility to directly deploy an executable model to run on a production system.

2.4.1 Execution Semantics

Among the approaches to define the execution semantics of a language presented in Section 2.2.3, translational and operational semantics are the best suited one to enable executability. In both cases, we consider that the semantics contains a data structure representing the *execution state*, which evolves during the execution. We first cover how this execution state is defined, and then present how both kinds of execution semantics are defined in the case of xDSLs.

[1] In the field of executable DSLs, the terms “model” and “program” are used interchangeably. In this thesis, we use the term “model”.

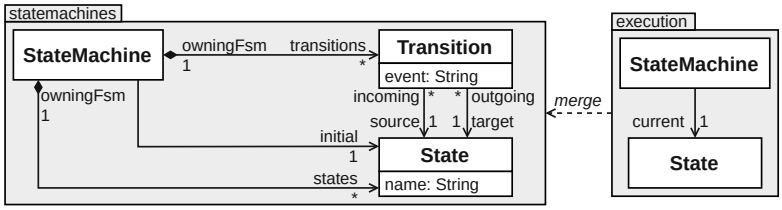


Figure 2.3: Package merge between the static and dynamic metamodels of the Statemachines DSL.

2.4.1.1 Execution State

The execution state of models conforming to an xDSL can be defined as an arbitrary complex structure, independent of the abstract syntax of the xDSL. However, if an xDSL features the static concept of *variables*, it seems convenient to link the dynamic concept of *values* from the execution state to the static concept of variable from the abstract syntax. For this reason, in many existing approaches, the execution state of xDSLs is defined by extending their abstract syntax, and by linking dynamic concepts to their corresponding static concepts.

For instance, in [HRV12], Hegedüs et al. define classes in a dynamic metamodel that may contain references to classes from the abstract syntax. Bandener et al. [BSE10] and Soden et al. [SE09] use a similar approach with a runtime metamodel. Mayerhofer et al. [MLWK13] proposes with the xMOF language to define configuration classes to extend the abstract syntax. A configuration class is a subclass of a class from the abstract syntax that introduces new properties specific to the execution state. Additional regular classes can be defined along these configuration classes to introduce execution-only concepts. Jézéquel et al. [JCB+13] provides similar facilities with the Kermeta language using aspect weaving. Similarly to the open class mechanism [CLCM00], an aspect can be defined to extend a class of the abstract syntax with new properties specific to the execution state. Additional classes can also be defined for execution-only concepts.

In essence, these approaches define the execution state by introducing new properties and/or new classes to the abstract syntax. We call *execution metamodel* the metamodel resulting from this extension. These approaches are very similar to an existing and well-known relationship between metamodels called package merge. This relationship was introduced in the Unified Modeling Language (UML) [Obj13a], and is also part of the MOF [Obj16]. A merge relationship between two metamodels declares the intent of merging classes of one metamodel into the other. The result of a merge is the set of all classes from both metamodels; if two classes have the same name, then they are combined in a class containing the properties from both originating classes. Package merge is conceptually very similar to the inheritance relationship between two classes, but as the metamodel level.

Figure 2.3 shows an example of package merge to extend the abstract syntax metamodel of the state machine DSL shown in Figure 2.1 with an execution metamodel specifying the execution state of conforming models. In this example, the execution

metamodel introduces a new relation between a StateMachine and its current State.

2.4.1.2 Operational vs. Translational Semantics

The execution of a model is the result of an in-place model transformation being performed on the model state. While in the case of operational semantics, this model transformation is performed by the execution semantics itself, the case of translational semantics is more complex. Indeed, having the execution semantics defined as a translational semantics for a DSL means that conforming (or source) models are not executed as-is but are first transformed into (target) models conforming to a target language. The target model is thus the one to be executed, through the execution semantics of the target language. The domain of this language (the target domain) might be completely different from the domain of the source language (the source domain), despite the latter being the one used by the domain expert. This defeats the purpose of executable metamodeling as it makes it difficult to interpret the execution from the perspective of the source domain.

To solve this issue, a few approaches have been proposed. In [HBRV10], Hegedus *et al.* propose to augment the translational semantics with back-annotations. This allows to translate the results of the execution (*i.e.,* the execution states) back to the source domain. Note that with this technique, due to a potential discrepancy between the abstraction level of the source and target languages, a single execution step in the source model can require multiple steps in the target model. For this reason, determining and detecting when to translate back to the source domain is a non-trivial task, but enables the use of dynamic analysis techniques at the appropriate domain. In [BW19], Bousse *et al.* propose a language engineering architecture to solve this problem. This architecture relies on the definition of the execution steps and states of models conforming to the source xDSL, and on a feedback manager able to translate the execution steps and states of the target model back to the source domain.

Regarding the implementation of xDSLs, translational semantics would result in a *compiler* while operational semantics would result in an *interpreter*. Back-annotation results in a mechanism similar to *debug symbols* used by interactive debuggers to visualize a current instruction or a stack from the perspective of a source model (e.g., Java code) while a target model is being executed (e.g., Java bytecode).

In the remainder of thesis, we only consider operational semantics for the definition of the execution semantics of xDSLs. More precisely, thereafter, the term xDSL only refers to xDSLs defined using operational semantics. However, note that our work can be directly adapted to translational semantics as long as a back-annotation mechanism is provided.

Listing 2.1 is a representative example of how the operational semantics of the state machines DSL can be defined with Xtend code. In this example, two execution rules are defined: the first one, handleEvent, is to be called when the state machine receives an event. It retrieves the first transition that can be fired upon the reception of the provided event, and then calls the second execution rule, fire, which performs the necessary updates to the execution state in response of the transition being fired.

```
def void handleEvent(StateMachine stateMachine, String event) {
    // find a transition to fire
    val Transition toFire = stateMachine.current.outgoing
        .findFirst[t|t.event.equals(event)]

    // fire it
    if (toFire != null) {
        toFire.fire
    }
}

def void fire(Transition transition) {
    // update the current state to the target of the fired transition
    transition.owningFsm.currentState = transition.target
}
```

Listing 2.1: Excerpt of the operational semantics for the state machines DSL (in Xtend).

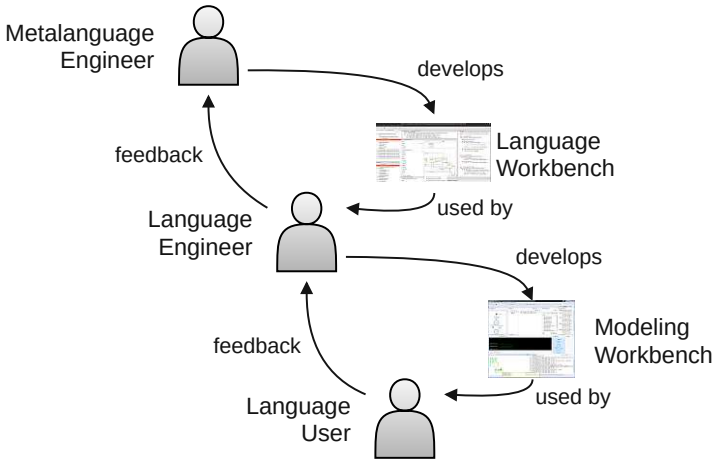


Figure 2.4: Roles in MDE.

2.4.2 Roles in Executable Metamodeling

The contributions in this thesis target users fulfilling different roles, depicted on Figure 2.4. We describe each of the three targeted roles thereafter.

Metalanguage Engineers Metalanguage engineers are the users that design metalanguages or adapt existing languages to be used as metalanguages for the definition of xDSLs. In addition, they develop the environment necessary to execute models, which we designate as execution engine, and possibly tooling that can be used with any language designed with their metalanguage (*e.g.,* debugger, tracing facilities).

Language Engineers/Metalanguage Users Language engineers design xDSLs using an array of metalanguages to define their abstract and concrete syntax, and their

execution semantics. These users also develop domain-specific tools for their languages, either from scratch or by reusing and/or extending generic tools provided by metalanguage engineers. The artifacts developed by metalanguage engineers are thus used by language engineers to define and update their xDSLs. In turn, language engineers provide feedback to metalanguage engineers, in particular on desired features to implement the specificities of the domain of particular xDSLs. Metalanguage engineers then attempt to provide building blocks allowing language engineers to implement the desired features into their xDSL.

Language Users Lastly, modelers are the users that design models conforming to xDSLs. Depending on how supported a given xDSL and the metalanguage used to define its execution semantics are, modelers will have access to a number of tools to aid them in their endeavor. The same kind of dialogue taking place between metalanguage engineers and metalanguage users also takes place between language engineers and language users. Modelers use the tools provided by language engineers and provide feedback on their specific use cases. Language engineers can then leverage this feedback to extend the expressivity of their language.

2.5 Behavioral Analysis

One of the motivation behind the definition and use of xDSLs is to be able to perform behavioral analysis on their conforming models, *i.e.,* study their behavior during their execution. Behavioral analysis is part of the long-standing V&V research field, that has seen an important amount of research for both hardware and software systems. An essential component of behavioral analysis is the definition of predicates on the evolution of a system during its execution. This requires appropriate operators dealing with the temporal aspect of an execution. To this end, various formalisms have been proposed, among them the Linear Temporal Logic (LTL) [Pnu77] and the Computation Tree Logic (CTL) [CE81]. LTL allows to define formulae that can be satisfied by infinite execution traces of systems. Through such formulae, one can assert, for example, that if a proposition p is true at a point in time, another proposition q will be true at a later point in time. CTL allows to define formulae on all the possible paths an execution can take. For instance, one can define a formula asserting that in every execution state, there is always a possibility to reach a certain state (also known as the reset property). Both of these formalisms have been and are being extensively used and studied. The most prominent techniques making use of LTL are *model checking* [CJGK+18] and *runtime monitoring*. CTL is incompatible with runtime monitoring but is used in model checking.

2.5.1 Model Checking

Model checking is an automatic technique for the validation of finite-state hardware and software systems. It consists in checking that a labeled transition system that models the system satisfies a temporal logic formula, expressed in LTL or CTL, specifying the desired

property. Thus, the verification procedure of these formulae consists in exhaustively searching the state space of the model of the system. This leads to the state explosion problem, which one must handle to successfully apply model checking. In the case of software model checking, this problem is exacerbated by the inherently unbounded constructs extensively used in the definition of software system, such as unbounded base types (*e.g.,* `float`, `string`), user-defined classes, dynamic method lookup, overloading, and absent source code for libraries, to name a few challenges. This means that some properties are too costly to verify with model checking techniques.

2.5.2 Runtime Monitoring

Another popular solution to ensure of the correctness of a system is *runtime monitoring*. Runtime monitoring consists in observing the internal state of a system during its execution to check whether the system satisfies or violates a correctness specification on this particular execution. This correctness specification is usually provided as a temporal property expressed in LTL [LS09], but other formalisms are supported, such as extended regular expressions and state machines.

Runtime monitoring can be split in two main categories: offline monitoring and online monitoring [CFAI17]. With *offline monitoring*, the monitor is executed independently from the monitored system, and works on the execution traces of the system instead of monitoring it during its execution. After a monitored system has executed, its execution trace is sent to the offline monitor which checks whether the monitored property is satisfied or violated on the execution trace.

Conversely, in *online monitoring*, the system is monitored while it is executing. This allows the monitor to incrementally check the execution of the system. Every time it observes a change in the internal state of the system, it checks whether the monitored property is satisfied or violated yet. Its verdict is based on the observed change and on the previously collected information.

In this thesis, we focus on a refined category of online monitoring called *synchronous monitoring* [CFAI17]. With synchronous monitoring, the runtime monitor is tightly coupled with the system. More precisely, the execution of the system is suspended whenever the monitor checks whether a received event violates the property or not. Compared to asynchronous monitoring, this allows timely (as opposed to late) detection of property violation or satisfaction, at the cost of performance decrease.

2.5.3 Complex Event Processing

While behavioral analysis relies on observing the model under execution, the granularity of these observations can be too fine to be practical for modelers. For instance, when the behavior of a model includes high numbers of iterations, observing arbitrary complex patterns of execution steps and states might be preferable to observing each single ones.

Complex Event Processing (CEP) is a form of Information Processing [CM12] that aims at defining and detecting such situations of interest —the so-called *complex events*— from a set of simpler events and their occurrences at runtime. This is done by defining

queries on event types, the data carried by their occurrence, and the temporal relationships between them. These queries are executed on event streams receiving the relevant event occurrences, including complex event occurrences. Thus, complex event processing allows to define arbitrarily deep event abstraction hierarchies [Luc02].

CHAPTER 3

State of the Art

In this chapter, we first survey the state of the art in light of the two challenges identified in Section 1.2. In more details, we look for tools and approaches that enable model interaction, and then survey techniques for offline and online dynamic analysis of executable models. Next, we investigate how language protocols (recent advances in the field of SLE) can be leveraged to design tooling applicable to a wide range of xDSLs. Continuing, we describe what can be seen as an emerging model execution protocol. Finally, we survey the available generic tools and approaches supporting the behavioral analysis of models conforming to xDSLs, and conclude on the contributions required to further extend this generic support.

3.1 Behavioral Analysis for xDSLs

In the field of executable metamodeling, the application of a number of behavioral analysis techniques in specific contexts has been the subject of a large amount of research. In light of the two identified challenges, we organize such approaches in 3 categories, as follows: (*i*) approaches enabling manual or programmatic model interaction, (*ii*) offline approaches improving the comprehension of model executions through their execution trace, and (*iii*) online approaches allowing the evaluation of temporal properties over model executions. In the remainder of this Section, we analyze each of these categories.

3.1.1 Runtime Domain-Specific Model Interaction

Often, xDSLs are used to model parts of systems that may interact with their *environment*, meaning that they can react to inputs and/or produce outputs. Such systems are called *reactive systems*. Accordingly, executable models representing these systems must be *reactive* as well for modelers to fully leverage the power of executable metamodeling. Indeed, designing parts of a reactive system through an xDSL does not allow to leverage the full potential of MDE if modelers cannot simulate the interactions that make this

system reactive. We divide approaches providing this interactivity in two categories: those working at the model level and those working at the language level.

3.1.1.1 Model-Level Approaches

A number of approaches allow to express reactivity on a model-by-model basis. This implies specifying what are the possible interactions for each model. Yakindu[1] is a state-of-the-art tool providing support for the definition of interaction interfaces for Statecharts models. Interaction interfaces identify input and output events for a model, and generate code allowing to both send corresponding event occurrences to the model, and listen to the emitted event occurrences.

In [DAH01], de Alfara *et al.* propose to augment software component interfaces with so-called *interface automata*. An interface automaton defines a protocol that implementations of the component interface must follow. This protocol is expressed in terms of the input and output actions declared by the interface. The authors provide the definition of a refinement relation, which can be used to verify that a software component correctly implements its interface with regard to its automaton.

In the area of simulation, the Functional Mockup Interface (FMI)[2] is an emerging standard to specify which variables are observed by the system, and which variables are exposed by the system. Simulation models are converted into executables called Functional Mockup Units (FMUs) which implement the standardized FMI, and each is accompanied with an XML model description of the interfaces of the unit. FMUs are mostly used for continuous models where time steps are performed, variables are set with initial values, and some variables may be observed during execution.

In essence, these approaches each provide a unified medium for interacting with particular sets of models (*e.g.,* Statecharts models, continuous models). However, interaction is both enabled at the model level and for a limited set of models only. This means that, to enable interaction for models belonging to an as-of-yet unsupported family of models (*e.g.,* models conforming to a new DSL), a new implementation is required. Furthermore, defining interaction-centric tools compatible with models conforming to different reactive DSLs is not possible if each DSL has its own way of enabling model interaction. This evidences the need for a unifying approach working at the language level.

3.1.1.2 Language-Level Approaches

Recent work showed the need for language-level approaches to define the possible interactions of conforming models with their environment. In [LDCM15], the authors advocate for the need of language behavioral interface for coordinating the execution of heterogeneous models and define one such interface as an extension of the abstract syntax of an FSM language. In [Dea16], the author similarly mentions behavioral language interfaces as a means to coordinate the execution of models conforming to heterogeneous languages. These works further evidence the need for a language-level, unifying approach

[1] https://www.itemis.com/en/yakindu/state-machine/

[2] http://fmi-standard.org

to model interaction. Furthermore, these works suggest such an approach should take the form of *language behavioral interfaces* allowing to define how models conforming to a language implementing such an interface exchange messages with other models, potentially conforming to other languages.

In [MDL+14], Meyers *et al.* presented the ProMoBox approach, which includes the generation of an *input metamodel* from the definition of a DSL, which can also be seen as a behavioral interface. However, this interface is only used for model checking, while our contribution is geared towards model execution. In [MDDV16], the same authors present an approach to augment an xDSL with reactive capabilities and generate a corresponding domain-specific *test language*. This requires to enrich the abstract syntax with event-related concepts and to accommodate for placeholder rules in the operational semantics, to be later replaced by calls to the test engine, which manages test cases and events. This effectively results in an altered version of the language, which cannot be decoupled from the test engine, for example to perform manual interaction at runtime, or to use conjointly with other tools. From these works, we can identify a need for a more modular approach to enable the use of a diverse array of dynamic analysis techniques on models conforming to reactive DSLs.

3.1.2 Offline Behavioral Analysis

Offline behavioral analysis consists in a set of techniques enabling the analysis of the behavior of a system outside of its execution, by considering its execution trace. These techniques are especially relevant when executing the model of a system of interest is particularly costly. Offline behavioral analysis techniques can be used to explore the semantic variations introduced by a modification of a part of the system, or to identify parts of the system performing sub-optimally. In the following, we divide these techniques in two categories: those based on the events observed during the execution of the system, and those based on the successive execution states traversed during the execution.

3.1.2.1 Event-Based Offline Behavioral Analysis

Event-based behavioral analysis is performed on execution traces recording events during an execution, which usually take the form of function or method calls. In [CZvD10], the authors highlight the difficulty to comprehend such execution traces as they are overloaded by noisy information.

Some techniques have been developed to reduce or summarize execution traces, helping in this matter. In [HLL06], Hamou-Lhadj *et al.* provide an algorithm for summarizing execution traces. The key idea is to remove from the execution trace all calls to so-called routines (*e.g.,* functions, methods) identified as *utilities* according to a criterion based on the number of calls to a given routine from distinct places in the program. In [BKD09], Bohnet *et al.* propose a classification of method calls allowing programmers to automatically prune method calls of little interest from their execution traces, complemented by the detection of repetitive sections of execution traces. The authors then provide a graph-like visualization allowing to identify cycles and offer a more

compact way of representing execution traces. More recently, Alimadadi *et al.* [AMP18] propose a high-level abstraction operator that detects recurring behavioral patterns and hierarchies thereof in sequences of events, with a tolerance for small variations in the patterns. Their algorithm is inspired from DNA sequence alignment algorithms used in bioinformatics.

Process mining is another approach that focuses on traces of events. Its main objective is to extract process-related information from event logs for providing information about actual processes [vdA11]. In [VDA12], discovery is mentioned as one of the main goals of process mining, *i.e.,* taking an event log as input and to produce a process model as output. Event log comparison techniques are also discussed in the realm of process mining [BvdA12].

3.1.2.2 State-Based Offline Dynamic Analysis

However, these techniques focus on the events contained in execution traces, and ignore the sequence of execution states through which the system went during the recorded execution. Yet, in the context of xDSLs, the link between the structure of a model and the sequence of rule calls resulting from its execution is less obvious than for programs written with GPLs. This is due to the fact that xDSLs are often declarative, possibly graphical languages, meaning that modelers do not consciously call the execution rules. Furthermore, the execution rules are hidden in the execution semantics, whose implementation details are not necessarily well-known to modelers. Conversely, what modelers are most familiar with is the *structure* of models, as it can generally be visualized and animated during the execution.

For this reason, several approaches rely on state-based execution traces (as opposed to event logs) to better understand the *semantic differences* between executable models. In [MRR11], Maoz *et al.* provide an algorithm producing witness execution traces that highlight the differences between two activity diagrams. This technique, realized specifically for activity diagrams, requires to devise an algorithm specialized for the xDSL of interest which combines its execution semantics and a kind of depth-first-search algorithm. However, in the general case, it might be more cost effective to record the execution trace of the two models, and compare these execution traces. This is the approach followed in [LMK14]. In this work, Langer *et al.* propose an approach that relies on the execution states contained in execution traces collected from the execution of models. Their work relies on dedicated *matching rules* to align pairs of traces in order to compute semantic differences, where a set of matching rules defines how traces should be meaningfully compared in the context of a given xDSL. However, this technique only eliminates noisy data for the time of the comparison, where further manipulation of the traces could be desirable for purposes other than comparison (*e.g.,* summarizing and visualization, cycle detection, etc.). This highlights the need for approaches offering reusable services within an ecosystem of tools with various purposes.

3.1.3 Online Behavioral Analysis

A widely adopted way to perform online dynamic analysis is to use runtime monitors defined on temporal properties. During the execution, runtime monitors check states and/or events to validate or invalidate the temporal property they monitor. However, in the context of xDSLs, using low-level temporal logic such as LTL or CTL poses some issues. Indeed, one of the main purposes of DSLs is to allow domain experts to take an active part in the design of a system. Asking such users to be sufficiently knowledgeable in LTL or CTL to check that their models behave as intended defeats this purpose. To alleviate the technicality of defining temporal properties in such a way, Dwyer *et al.* propose in [DAC98] a set of Property Specification Patterns (PSPs) to facilitate the definition of temporal properties by proposing an alternative to these low-level logics. The PSPs are composed of temporal patterns expressing for instance that a specific event must never be observed during a given interval, or that a specific event must always precede another. An additional constraint can then be imposed on these pattern, under the form of scopes, dictating when during the execution a pattern must be observed (*e.g.,* between two specific events, before a given event, etc.).

In the context of SLE, this foundational work was instrumental in the design of several languages allowing modelers with little to no knowledge of LTL or CTL to define runtime monitors. For instance, Li *et al.* [LJH06] propose a language directly based on the PSPs to define constraints for web service interactions. The authors provide a translational semantics to finite state automata for each scope and temporal pattern, allowing to obtain runtime monitors from the constraints defined using their language. Barnawi *et al.* [BAE+15] propose a runtime monitoring approach for BPMN. In this work, the authors rely on CEP to check for violations of compliance patterns, which are derived from the PSPs. Other approaches instead leverage the PSPs for validation purposes. For instance, Simmonds *et al.* [SGC+09] propose a property specification language based on Sequence Diagrams (SD) to express conversations between web services. They provide a formal semantics for their SD language, which allows them to derive runtime monitors from the properties defined with the language. They evaluate the expressiveness of the language by using it to reproduce the PSPs. While these approaches highlight the suitability of the PSPs for expressing temporal properties, they are specific to particular domains and cannot be directly reused for other xDSLs.

Taking a language-generic direction on behavioral analysis, Meyers *et al.* [MDL+14, MDDV16] propose, as part of their ProMoBox approach (mentioned in Section 3.1.1.2), to generate a set of languages from an existing DSL, including a property language inspired from the PSPs, and a testing language allowing to define test cases for reactive models. They supply an accompanying testing engine that allows to execute the test cases defined with the generated language. However, to use this testing engine, specific adaptation must be performed on the execution semantics of the DSL. This hampers the interoperability of the testing engine with other tools. In [DT16], Drey *et al.* establish a list of requirements for the integration of runtime monitors with the execution semantics of xDSLs, and propose a design pattern fulfilling these requirements that allows such an integration with xDSLs whose execution semantics is defined according to the visitor

design pattern. This work departs from PSPs-inspired approaches and instead enables runtime monitors defined in a programmatic way (*e.g.,* written in Java) to be composed with the execution semantics of the DSL. For this reason, this approach is mostly geared toward language engineers and modelers with a technical background in computer science.

3.2 Language Protocols and Software Language Engineering

Among recent advances in SLE are a new kind of protocols: language protocols. Stemming from a need to shift from heavy, language-specific IDEs to lightweight, modular and possibly web-based IDEs, these protocols, namely the Language Server Protocol[3] (LSP) and the Debug Adapter Protocol[4] (DAP), are dedicated to core functionalities expected by language users: editing and debugging. These language protocols enable a shift of the intelligence from development tools to interchangeable backends, which in turn allows the provision of lightweight, generic IDEs that can be configured with multiple language backends, according to the needs of the user. In the remainder of this Section, we first explore LSP and DAP in more details, before covering their influence on SLE practices, and in particular in executable metamodeling.

3.2.1 LSP and DAP

The LSP aims at standardizing the messages exchanged between tools like code editors and IDEs, and servers providing language-specific editing features, such as auto-completion, syntax highlighting, refactoring, jump to declaration, and so on. In an LSP-based architecture, clients provide a language-agnostic front-end text editing support, while servers provide the language-specific support, which mainly consists of the static semantics of the language. This architecture offers developers the freedom to choose the most suitable technology to implement the client editor and the language server independently of each other. Currently, LSP only supports text documents (*i.e.,* languages with a textual concrete syntax) through a set of primitives that can fit to any text editor, such as the currently open text document, the current position of the cursor or the range of selected characters. This allows clients to connect to different language servers seamlessly, thereby easily providing editing support for new languages.

Where LSP is focused on the static semantics of languages, another protocol, the DAP, deals with the execution semantics of languages. In more details, DAP is a standardized protocol specifying how development tools such as IDEs communicate with concrete, language-specific debuggers providing debugging facilities such as stepping operators (*e.g.,* step into, step over) and the inspection of instance values. Similarly to an LSP-based architecture, in a DAP-based architecture, development tools are language-agnostic and communicate with language-specific debuggers in charge of orchestrating the execution. DAP provides a set of debugging and runtime primitives designed to support most of

[3] `https://microsoft.github.io/language-server-protocol/`
[4] `https://microsoft.github.io/debug-adapter-protocol/`

the existing debuggers. Such primitives include events indicating when the execution has stopped due to a breakpoint, requests to restart the execution or to read bytes from memory at a given location. Similar debugging protocols are available for specific languages, such as the Chrome DevTools protocol for Javascript[5], the Java Debug Wire Protocol (JDWP)[6], and the GDB/Machine Interface for C and C++[7].

3.2.2 Language Protocols in MDE

The emergence of LSP, due to its clean separation between the editor and the language server, is an advance that has a high impact on the useability of DSLs [Bün19]. Indeed, from the perspective of the modeler, LSP enables the use of a language in a lightweight, possibly web-based editor, without going through an extensive installation process, both of which facilitate the adoption and use of DSLs by users without a technical background in computer science. From the perspective of the language engineer, the provision of a language server for their DSL can be automated at the metalanguage level, as is already the case for Xtext for instance. This means that it does not incur additional development costs for the language engineer. Yet, LSP only provides support for textual languages, which means that its applicability in MDE remains partial, as it leaves out graphical modeling languages. This prompted the provision of several approaches aiming to adapt LSP for graphical modeling [REIWC18].

Concerning the adoption of DAP in MDE, besides GPLs such as Java, C# or C++, no metalanguage dedicated to the definition of the execution semantics of xDSLs comes with a DAP-compliant debugging back-end yet. However, there is existing work investigating how to decouple runtime services from the metaprogramming approach used to implement the execution semantics of xDSLs. As such runtime services include debugging, this work can be likened to DAP. It is described in the next section.

3.3 An Emerging Model Execution Protocol

Recently, Bousse *et al.* [BLC+18] proposed an execution engine interface and its accompanying execution listener interface, designed to observe and interrupt the execution of models. On top of this execution listener interface, the authors define an interactive omniscient debugging interface for xDSLs. Together, the execution engine and interactive debugging interfaces can be considered as a subset of DAP that decouples execution from interactive debugging. In particular, the execution engine and execution listener interface constitute an emerging model execution protocol. Implementing the execution engine interface yields two main benefits for metalanguage-specific execution engines. First, the complying execution engines are able to run any model conforming to xDSLs whose

[5] Chrome DevTools Protocol, Google,
https://chromedevtools.github.io/devtools-protocol/

[6] Java Debug Wire Protocol, Oracle Inc.,
https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/jdwp-spec.html

[7] GDB/MI Interface, The GNU Project Debugger,
https://sourceware.org/gdb/onlinedocs/gdb/GDB002fMI.html

execution semantics is defined with the metalanguage to which the engine is dedicated. Second, developing tools implementing the execution listener interface guarantees their reusability across all complying execution engines, and thus across all xDSLs supported by such engines (as long as the tools are not domain-specific themselves). Thereby, the execution engine interface allows to decouple external tools offering runtime services from the metalanguages used to define the execution semantics of xDSLs. This in turn fosters the reuse of such tools across xDSLs defined with different metalanguages.

In more details, at the heart of the execution engine interface is the concept of interrupting the execution of models on *consistent* execution states. This in turn allows external tools to safely observe the model state and possibly control the execution through the execution engine. Safe interruption of model execution is achieved by annotating transformation rules from the execution semantics as *step rules*. When a rule is annotated, it means that the execution can be interrupted both at the beginning and at the end of the application of the rule. Accordingly, execution engines complying to the interface feature a system of *execution listeners*. Such listeners can register and unregister from a given execution engine. Each time the execution is interrupted, a complying execution engine will send the corresponding notifications to all its registered listeners. This allows the definition of external tools as execution listeners, and constitutes an important step toward enabling generic behavioral analysis for xDSLs, as it provides a unified way to both execute models, and observe and control their execution. We cover this topic in the next section.

3.4 Generic Behavioral Analysis for xDSLs

From the way their abstract and concrete syntaxes are defined, down to the metaprogramming approach used to define their execution semantics, which can itself take various forms (*e.g.,* operational, translational, etc.), xDSLs come in many shapes and forms. Hence, providing behavioral analysis facilities for a particular DSL often requires to develop new tooling from scratch, or at least to adapt existing tooling to the specificities of this DSL. This is a tedious and error-prone task, that must be repeated for each new DSL to obtain a functioning modeling environment (*i.e.,* a modeling workbench). For this reason, a number of generic facilities for behavioral analysis have been proposed based on the emerging model execution protocol mentioned in Section 3.3. These facilities provide or facilitate the provision of tooling available out of the box for any xDSLs whose execution semantics can be run by a compliant execution engine. A tour of these facilities is given thereafter.

3.4.1 Model Animation

A first use of model execution listeners is to enable modelers to observe how the execution of their models unfolds by animating their representation [BDV+16]. This is a natural extension for graphical modeling workbenches, as it expands the benefits of graphical concrete syntax for static models to dynamic models by providing a better visualization of

the impact of model changes on the execution to modelers. To achieve this, the graphical concrete syntax of the DSL must be extended to cover its dynamic metamodel, *i.e.,* the parts that will animate during the execution. With such an extension defined, graphical representations can be defined for executable models. Animating these representations then consists in observing the execution of their corresponding model and refreshing them every time it reaches a consistent execution state.

3.4.2 Model Debugging

In [BLC+18], Bousse *et al.* leverage execution listeners to provide a generic model debugger supporting classic debugging operations such as pausing the execution and controlling how the execution unfolds through stepping operators (*e.g.,* step into, step over, etc.). This is done through an execution listener able to suspend the execution and wait for debugging commands from the modeler, under the form of the aforementioned stepping operators, or pause/resume commands. As this generic debugger receives notifications when execution steps begin or end, it is able to reproduce the behavior of the step into (by suspending the execution on the next step), step over (by suspending the execution when the next started step ends) and step return (by suspending the execution when the current step ends) operators.

3.4.3 Model Tracing

Leveraging this, in [BMCB17], Bousse *et al.* propose an approach to efficiently record execution traces from executed models. In this work, a trace recorder is defined as an execution listener that constructs efficient execution traces during the execution of models. These traces are complete as they record all model states, as well as all the execution steps (*i.e.,* the execution events) leading from one model state to the next. An offline use for these execution traces is to perform semantic differencing between distinct versions of a given model, to precisely pinpoint the differences between two versions in terms of semantics. In addition, an online use for these execution traces is explored in [BLC+18]: omniscient debugging for xDSLs. This is achieved by leveraging the tracing facilities to restore running models to their past execution states, thereby enabling generic omniscient debugging for xDSLs. On top of these low-level state restoring facilities, several backward state navigation operators are formalized, mirroring the classic forward stepping operator that can be found in most debuggers. The definition of these operators leverage the execution step hierarchy recorded in the execution trace to determine in which execution state and in front of which execution step the model execution should be restored. Note that these facilities only offer online replayability and do not allow to explore different execution path, for instance by changing a dynamic value of the model before resuming the execution.

3.5 Summary and Overview of the Contributions

The emerging model execution protocol paves the way for better generic support for online behavioral analysis of xDSLs, and the already existing support for model animation, tracing and debugging already provides solid foundations. Yet, in the case of reactive DSLs, the potential of such tools cannot be fully leveraged unless the modeler is able to explore the reactive aspect of these DSLs, either manually or programmatically. This in turn requires the use of tools dedicated to reactive DSLs. To enable the definition of such tools in a generic way, the reactive aspects of these DSLs, *i.e.,* the interactions conforming models can have with their environment, must be defined in an explicit and unified way. As seen previously, several works call for the use of behavioral language interface to fulfill this need.

Another concern which is orthogonal yet complementary to the reactive aspects of xDSLs is the genericity of tools dedicated to the behavioral analysis of their conforming models. Indeed, while numerous tools exist for this purpose for specific GPLs and xDSLs, few approaches conjugate behavioral analysis of executable models conforming to *any* xDSL on one hand and the use of the domain concepts of these models on the other hand. Yet, such approaches are valuable as they enable the provision of tools that are both suitable for use by modelers, and reusable across xDSLs. The concept of model execution protocol plays a central role in the provision of such approaches, as it enables a communication from an undergoing model execution to its potential observers, an essential part of behavioral analysis tools.

In this context, as a first contribution of this thesis, we propose a new metalanguage, rich with its abstract syntax and operational semantics, to define the behavioral types of reactive DSLs under the form of behavioral language interfaces. Then, as a complement to the preexistent generic model tracing facilities, we propose a second contribution: a set of trace comprehension operator allowing to manipulate, analyze and compare execution traces. The operators constitute an algebra enabling the automatic transformation of execution traces before comparing them or representing them as a state graph. Finally, we propose an accessible temporal property language allowing to define domain-specific runtime monitors with the intuitive temporal patterns known as the PSPs. The runtime monitors derived from the temporal properties can serve as test oracle, conditions for breakpoints and other online behavioral analysis activities.

CHAPTER 4

Behavioral Interfaces for Executable DSLs

While behavioral models are often used to represent reactive systems, they are seldom reactive themselves. Moreover, those that are reactive only offer interaction mechanisms that are specific to the xDSL to which they conform. This means that modelers cannot expect to always benefit from interaction-centric tool support to help them verify that their models behave as expected. This also means that the few existing interaction-centric tools cannot be easily reused from one xDSL to another, thereby forcing language engineers to implement such tools anew or to adapt existing ones to provide adequate tool support to modelers. Therefore, a prerequisite to foster the emergence of an ecosystem of interoperable and generic tools dedicated to the behavioral analysis of reactive models is the provision of unified interaction facilities across xDSLs.

In this chapter, we propose to achieve this by introducing a new metalanguage for extending the specifications of xDSLs with their behavioral types. This metalanguage allows language engineers to define both explicit behavioral language interfaces, and the implementation and subtyping relationships between such behavioral language interfaces and xDSLs. This way, language engineers define the behavioral type of xDSLs, and expose the interactions that conforming models can have with their environment, effectively turning their DSL into a reactive one.

Language engineers can then leverage these explicit behavioral types to design interaction-centric tools that are either generic, or specific to the domain of particular behavioral types. This in turn both extends the scope of xDSLs supported by existing tools for behavioral analysis to include reactive DSLs, and contributes to enriching this ecosystem of tools with interaction-centric ones.

The work presented in this chapter is the subject of a publication in the International Journal on Software and Systems Modeling [LBW+20].

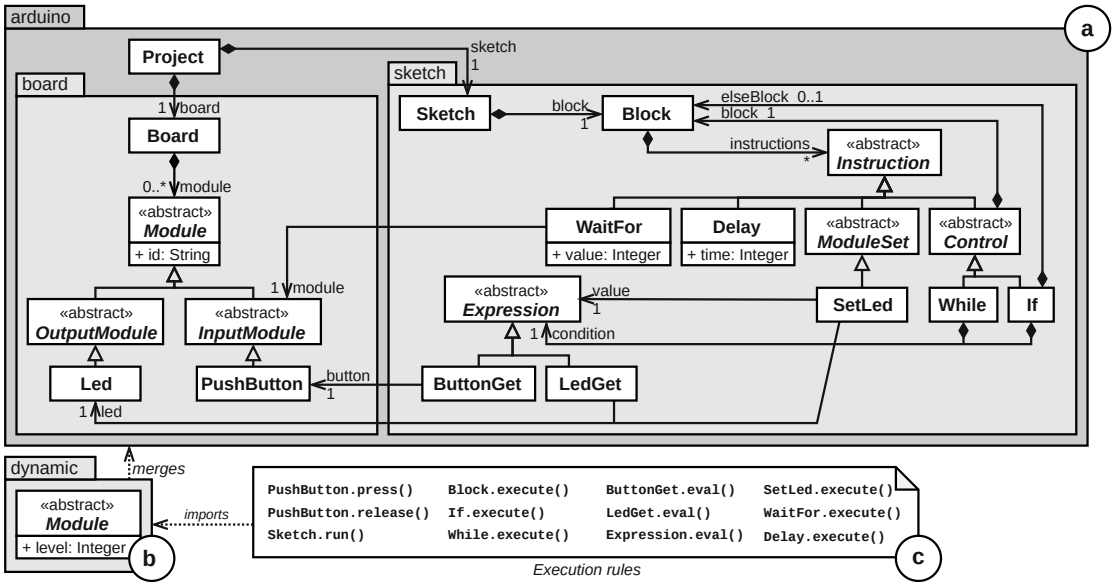


Figure 4.1: Arduino executable DSL definition.

4.1 Motivation

In this section, we precisely scope the xDSLs considered in our approach and then motivate our approach using an illustrative example.

4.1.1 Considered Executable DSLs

In this chapter, we focus on DSLs where (1) the abstract syntax is provided as a *metamodel* defined using a metamodeling language (*e.g.,* MOF [Obj16] or Ecore [SBPM08]) and (2) the execution semantics is provided as an operational semantics (*i.e.,* an interpreter).

In addition to the metamodel defining its abstract syntax, an xDSL can expose several structural language interfaces constituting its available model types [DCB+17, GCD+12, SJ07]. These model types define a set of metaclasses and structural features that are guaranteed to be present in the metamodel constituting the abstract syntax of the DSL, and thus supported by its conforming models.

The considered operational semantics are those composed of a data structure representing the *model state* and a set of *execution rules* altering this model state.

Definition 1. We define the operational semantics of an xDSL as a tuple $\langle DM, ER \rangle$ where DM is its dynamic metamodel, and ER its set of execution rules.

The model state is defined in an *execution metamodel* extending the abstract syntax metamodel using a non-intrusive extension mechanism, such as *package merge* [Obj13a] or *aspect weaving* [JCB+13]. The execution rules perform an in-place endogenous transformation on this model state, resulting in the execution of the model. While the operational

semantics can handle time (*e.g.,* through a central clock), the proposed approach is time-agnostic.

Definition 2. We define an xDSL as a tuple $\langle AS, OS \rangle$ where AS is its abstract syntax, and OS its operational semantics.

Figure 4.1 shows the definition of the Arduino xDSL, which will be used as a running example throughout this chapter. The abstract syntax of the DSL is defined as a metamodel (**a** in Figure 4.1). A Project contains a Board and a Sketch. The `board` of the project represents the physical Arduino board on which the `sketch` of the project is executed. A Board contains Modules, which have an `id` attribute. A Module can either be an OutputModule, such as a Led, or an InputModule, such as a PushButton. Being a program to be executed on a Board, a Sketch contains a Block of Instructions that can be Control, ModuleSet, Delay or WaitFor instructions. Control instructions come in the usual forms of If and While instructions. They contain a Block (or potentially two for the If instruction) and a `condition` in the form of an Expression. For the sake of brevity, the whole class diagram of Expression is not shown here, except for the ButtonGet and LedGet classes, which respectively point to a PushButton and a Led. The ModuleSet class is further specialized for each OutputModule: here, SetLed is a ModuleSet instruction for Led modules, setting the `level` attribute of its associated Led to the result of the evaluation of its `value` expression. The Delay instruction suspends the execution for the specified amount of milliseconds. The WaitFor instruction points to an InputModule and suspends the execution until the `level` of the referenced module reaches the provided `value`.

The bottom part of Figure 4.1 shows the two parts of the operational semantics of the Arduino DSL. The execution metamodel (**b** in Figure 4.1) extends the Module class with the `level` integer attribute, indicating the logic level of the signal transiting between a Module and its containing Board. For Led modules, the `level` represents whether the LED is lit or not, whereas for PushButton modules, it indicates whether the button is currently pressed or not. The execution rules (**c** in Figure 4.1) import this execution metamodel and consist of model transformations defining how the state of a running model is altered. In the case of the Arduino DSL, only the `level` attributes of Led and PushButton elements can be changed, either by the SetLed.execute rule for Led modules, or PushButton.press and PushButton.release for PushButton modules. As the execution semantics of the Arduino DSL is implemented as a visitor, the scheduling of the execution rules is determined internally. The Sketch.run rule is the entry point rule of the DSL: it starts the model transformation resulting in the execution of conforming models. To this end, it calls the Block.execute rule of its contained Block, thereby starting the visit of the containment tree of the running model. The Block.execute rule sequentially calls the execute rule of the `instructions` it contains. The If.execute rule calls the Block.execute rule on its `block` (resp. `elseBlock`), if its `condition` evaluates to `true` (resp. `false`). Similarly, the While.execute rule calls the Block.execute rule on its `block`, while its `condition` evaluates to `true`. The remaining execution rules are dedicated to the the implementation of the behavior of the SetLed instruction and of the waiting

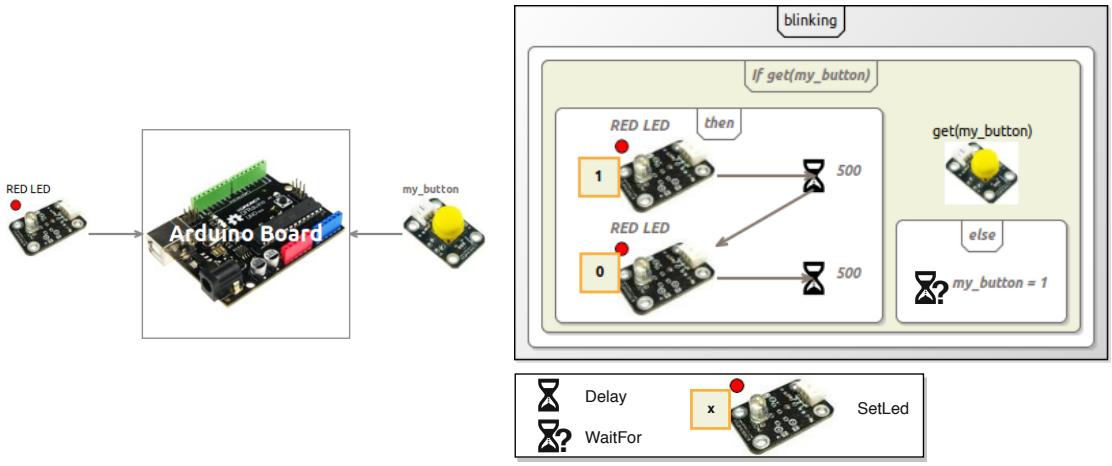


Figure 4.2: Example Arduino model.

mechanism of the Delay and the WaitFor instructions. The complete definition of the DSL is available on Github[1].

4.1.2 Motivation & Requirements

Figure 4.2 shows an example model conforming to the Arduino DSL. This model represents an Arduino circuit with one button and one LED, where the LED blinks while the button is pressed, and remains off otherwise.

If we consider an execution of this model where the button is pressed in the initial state, and remains pressed in all states, we observe that the LED blinks as defined in the model. However, such an execution scenario does not show whether the LED eventually stops blinking when the button is released, or more generally how the LED behaves with different scenarios of button pressings. To test more complex execution scenarios, the modeler must be able to change the state of the button *during* the execution of the model.

Since our operational semantics does not provide any explicit way to interact with a running model, one possibility is for the modeler to directly modify the state of the model during its execution, effectively resulting in a form of stimulus. Figure 4.3 shows an execution trace where two changes ($my\_button.level = 1$ and $my\_button.level = 0$) are made during the execution, the first changing the `level` attribute of the PushButton to 1, and the second changing it to 0. The modeler can thus effectively observe that the LED not only blinks when the button is pressed, but also stops blinking when the button is released. While this solution does allow the execution of specific scenarios, it has several limitations, which can be divided in two categories.

The first category of limitations relates to the way the possible stimuli that can be sent to models are defined. Manipulating these stimuli as model changes is both

[1] https://github.com/tetrabox/examples-behavioral-interface

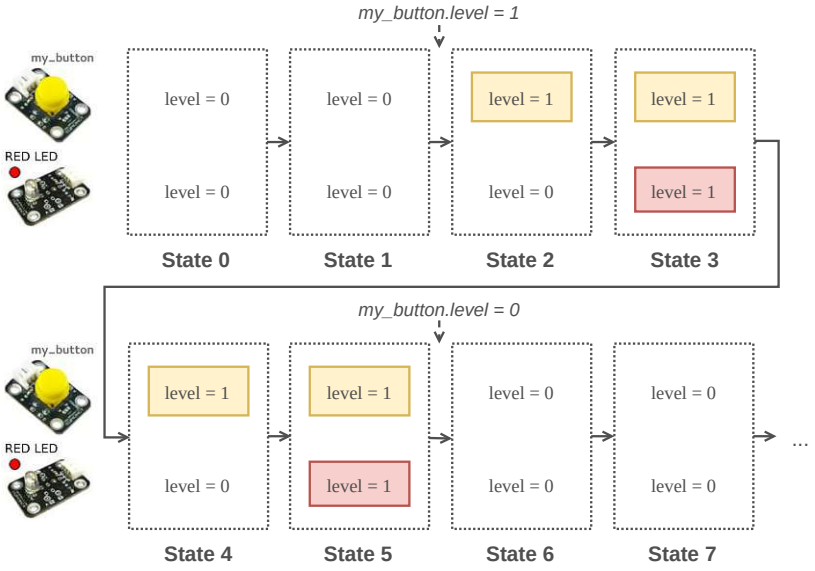


Figure 4.3: Execution of the Arduino model (Figure 4.2) where the PushButton is only pressed between states 2 and 5.

a cumbersome and error-prone process for modelers. For instance, issuing the model changes corresponding to a given stimuli and interpreting observed model changes both necessitate extensive knowledge of the operational semantics of the DSL, which modelers are not assumed to have. In addition, it can result in unsound behavior with regards to the operational semantics of the DSL. For example, the semantics of a language may restrict the subset of the execution state of a conforming model that can be affected by an external stimulus (*e.g.,* the status of an InputModule of an Arduino board), while the remainder of the state should only be affected by the inner operational semantics (*e.g.,* the status of an OutputModule). One way to circumvent these problems is to provide a clearly defined way for language engineers to define the *behavioral types* of xDSLs. The purpose of these behavioral types is to expose the domain-specific stimuli of an xDSL as first-class entities that are part of the language definition. A widely adopted approach for the reification of stimuli types is to consider such stimuli as occurrences of well-defined *events* (*i.e.,* their type). This would allow language engineers to attribute a type to the stimuli received and sent by models and thus facilitate both their manipulation by modelers and *sound* interaction with models. For example, in Figure 4.3, the two model changes made by the stimuli during the execution correspond to a particular button being pressed and released. From the perspective of the modeler, this is not strikingly clear. The language engineer can improve this by defining two types for these stimuli: the *pressed* and *released* stimuli types, which both convey more domain semantics than low-level model changes. From these limitations, we can define a first requirement for the approach as follows:

Req. 1 "Provide an explicit and unified way to *define* the behavioral type of an xDSL, *i.e.,* how to soundly interact with any model conforming to the DSL".

The second category of limitations relates to the way stimuli are sent to and received from the running model, and to the soundness of the resulting behavior. For example, the operational semantics may only allow some stimuli to affect the execution state at certain points in time or when it is in a specific state. Since arbitrary transformations may break these constraints, it appears important to control which and when changes of the model state are allowed, *e.g.,* by only allowing specific execution rules to be called in reaction to stimuli and under specific circumstances. Another limitation is related to the concurrent execution of multiple transformations on the same model state, which can quickly lead to undefined behaviors when some values are simultaneously changed externally and by the operational semantics. It appears therefore important to also control when stimuli-triggered changes can be applied to the model state, *e.g.,* by delaying their application until the currently executing rule yields back control if it is a run-to-completion rule call. Moreover, xDSLs are only as useful as the richness of the ecosystem of tools defined for them. One limitation of the solution proposed in Figure 4.3 is that it does not provide a unified way to send stimuli to the model. This hampers the definition of tools interacting with running models. Conversely, this solution does not provide a clear way for the model to emit stimuli of its own towards external tools. Both cases thus require ad hoc techniques to either inspect observable parts of the state of a running model (*e.g.,* to detect when a LED is switched on or off), or send stimuli to running models. From these limitations, we can define a second requirement on the approach as follows:

Req. 2 "Provide a unified way to *interact* soundly with models conforming to xDSLs implementing one or several behavioral types".

However, in model-driven engineering, parts of a system can be modeled using various DSLs fitting different needs such as model checking, simulation, animation, and so on. Due to their individual particularities, such DSLs potentially accept and expose different events. This prevents modelers from using the same events to interact with models conforming to different DSLs despite representing the same part of the system. One way to answer this problem is to allow language engineers to define a set of events abstracting the various events defined for each of these DSLs. From there, language engineers can define how this set of abstract events maps to other sets of events, effectively defining overlapping event abstraction hierarchies. Using these mappings at runtime to translate event occurrences would enable modelers to interact with models conforming to any of the covered DSLs through this single set of abstract events. In addition, language engineers would foster the emergence of families of xDSLs supporting a shared set of abstract events. For example, the Arduino model shown in Figure 4.2 could be the realization of a specification available as a model conforming to a State Machine DSL. By defining how events supported by State Machine models can be mapped to events supported by Arduino models, a language engineer would enable interaction with both kinds of models

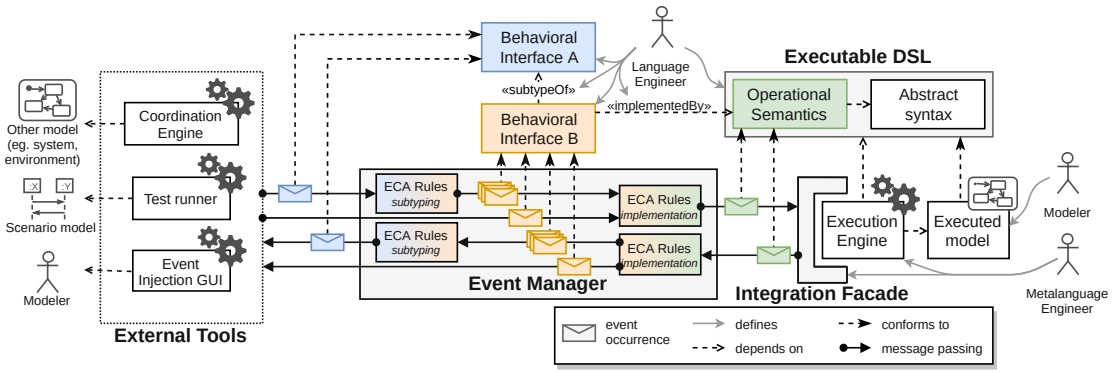


Figure 4.4: Overview of the approach.

using the same set of events. From this scenario, a third and last requirement on the approach can be formulated:

Req. 3 "Support the definition of overlapping event abstraction hierarchies for xDSLs".

To support these scenarios, we propose a new metalanguage to specify in a unified way the behavioral types of xDSLs under the form of *behavioral interfaces*, thereby fulfilling **Req. 1**. Implementation relationships can then be established between xDSLs and their implemented behavioral interfaces. At runtime, these relationships configure a generic event manager to enable safe interaction with the running model, while keeping a clear separation between the implementation of an xDSL and its interfaces. In turn, this event manager exposes the available behavioral interfaces of the running model, thereby enabling the definition of generic interaction-centric tools and fulfilling **Req. 2**. Finally, we introduce subtyping relationships between behavioral interfaces, allowing to define event abstraction hierarchies and thus fulfill **Req. 3**. We provide an overview of the complete approach in the following section.

4.2 Approach Overview

We provide in this section an overview on both the design of the approach and its envisioned use by developers.

4.2.1 Design Overview

Figure 4.4 depicts an overview of the proposed approach. On the top right corner, the xDSL complies with the definition given in Section 4.1.1. As such it contains an abstract syntax as a metamodel and an operational semantics with both a set of execution rules and a data structure defining the model state. On the bottom right corner is shown a running model whose static content conforms to the abstract syntax, and whose dynamic state conforms to the execution metamodel. Next to it, the *execution engine* is able both to apply any execution rule of the operational semantics on the running model,

and to notify execution observers when execution rules are applied. Such an execution engine is based on our previous work on decoupling operational semantics from execution observers [BDV+16, BLC+18].

On the left, examples of external tools that require interacting with the running model are represented.

- A coordination engine managing the communication with other models —representing some part of the environment or other parts of the system— during an execution.
- A test runner executing a specific scenario model, alternating between sending sequences of stimuli to the model and checking whether a proper sequence of stimuli are received from the model in return.
- An event injection GUI complementing the classic stepping operators of interactive debuggers (*e.g.,* step into, step over) with the capability to manually send domain-specific stimuli to the running model, and to observe the stimuli produced in reaction.

For language engineers, developing such tools as needed for each new xDSL is both a tedious and error-prone process. Providing to language engineers a unified way to define the possible interactions with models conforming to any xDSL would allow them to define generic tools instead. To achieve this for any xDSL included in our scope (see Section 5.1), we introduce *behavioral interfaces* as behavioral types of xDSLs.

Using the proposed approach, language engineers can define behavioral interfaces, or reuse existing ones, to type their xDSLs based on the runtime interaction capabilities offered by their conforming models. When external tools discover the behavioral interfaces of an xDSL, they are informed of the kind and form of stimuli that can be emitted and/or received by conforming models. This then allows modelers and other models to interact with conforming models through these tools.

Such an interface consists of an *event metamodel* that defines the exact set of *events* that are relevant to the interface purpose and/or domain. First, this means defining the events whose occurrences can be *accepted* or *exposed* by models conforming to xDSLs typed by the behavioral interface. Second, this means specifying the nature and structure of the data that can potentially be carried by occurrences of these events.

As shown in Figure 4.4 by the dependency between behavioral interface B and the operational semantics, language engineers can type their xDSL by a given behavioral interface by providing an *implementation relationship* between the interface and the xDSL, which amounts to nominal typing [PB02]. This implementation relationship describes how the xDSL provides the interaction capabilities that are expected of a language typed by the behavioral interface. In practice, this is done by detailing how occurrences of the events defined in the interface translate in term of actual behavior, and vice-versa. As a supplementary contribution, we provide a systematic way to generate the behavioral interface implemented by an xDSL, as well as the implementation relationship between them.

In addition, we introduce nominal subtyping [PB02] for behavioral interfaces in the form of *subtyping relationships*, illustrated between behavioral interfaces A and B on

Figure 4.4. By defining a subtyping relationship between two behavioral interfaces, language engineers designate one interface as the subtype and the other as the supertype. A subtyping relationship then dictates what patterns of accepted (resp. exposed) event occurrences from the supertype (resp. subtype) translate to which event occurrences from the subtype (resp. supertype), in what order and carrying what data. Using subtyping relationships, language engineers can capitalize on the behavioral similarities of different xDSLs to define tools that are both specific to these similarities and reusable across any DSL exhibiting them.

Both implementation and subtyping relationships are realized through Event-Condition-Action (ECA) rules, as shown on Figure 4.4. These rules are triggered when a pattern of event occurrences —the event part of the rule— from one side of the relationship is observed, given that their associated condition (*e.g.,* an OCL query, a Java predicate) is satisfied. When triggered, their action part translate the observed event occurrences into new event occurrences belonging to the other side of the relationship. As shown on the figure, ECA rules are managed by the event manager, a component able to route event occurrences from and to the various ECA rules, the execution engine and the external tools.

Finally, we propose an integration facade for the event manager that acts as an intermediary between the execution engine and the aforementioned event manager. While this integration facade is specific to the metalanguage that is used to define the operational semantics of xDSLs, the rest of the approach is agnostic to any such metalanguage. Furthermore, defining this facade for a metalanguage enables the approach for any xDSL whose operational semantics is defined with this metalanguage.

In the end, the three main constituents of the approach —the behavioral interface metamodel, the relationships and the event manager— form a metalanguage to extend an xDSL with an event handling component. The behavioral interface metamodel is used to define the abstract syntax of such language extensions, while the relationships define their operational semantics. Models conforming to this language extension are occurrences of events defined in behavioral interfaces. At runtime, the event manager acts as the engine executing the operational semantics of the language extension (*i.e.,* manage event occurrences and relationships), thereby forming an interpreter for the language extension.

4.2.2 Usage Overview

In this chapter, we distinguish three kinds of users for the approach: metalanguage engineers, language engineers and modelers. We describe hereafter each of these kinds, which are represented on Figure 4.4.

Metalanguage engineers are the users that design metalanguages or adapt existing languages to be used as metalanguages to define xDSLs. In addition, they develop the environment necessary to execute models, which we designate as execution engine, and possibly tooling that is generic to any language designed with their metalanguage (*e.g.,* debugger, tracing facilities). With the proposed approach, they can provide an integration facade for their execution engine to enable any xDSL based on their metalanguage and implementing a behavioral interface to use existing generic interactive

tools that work with any behavioral interface. This has a cost for metalanguage engineers, but we believe that, as their role is to provide facilities to create new languages, they have a great incentive to add such a facade. However, it is possible that in some cases the person with the role of language engineer can temporarily take on the role of metalanguage engineer to provide the integration facade through a pull request or a similar process. But in that case, the correctness of the integration facade still has to be assessed by a metalanguage engineer.

Language engineers are the users that design xDSLs using a metalanguage to define their execution semantics. These users also develop domain-specific tools for their languages, either from scratch or by reusing and/or extending generic tools provided by metalanguage engineers. With the proposed approach, they can provide or reuse behavioral interfaces as well as implementation and subtyping relationships between these interfaces and their xDSLs. In addition, language engineers can provide or reuse interactive tools that are specific to the behavioral interfaces implemented by their xDSLs. To enable this, they need to learn the proposed metalanguage and how to define relationships, but we believe that the benefits outweigh the costs as soon as this enables the direct reuse of even a small set of tools.

Lastly, modelers are the users designing models conforming to xDSLs. Depending on how supported a given xDSL and the metalanguage used to define its execution semantics are, modelers will have access to a number of tools to aid them in their endeavor. With the proposed approach, modelers get access to any generic interactive tooling, as well as any tooling specific to a behavioral interface implemented, either directly or transitively, for the xDSLs they use.

In what follows, we first provide a specification for behavioral interfaces, and implementation and subtyping relationships in Section 4.3. Then, we detail one possible strategy to realize the proposed approach in Section 4.4.

4.3 Behavioral Interface & Relationships

In this section, we first specify what are behavioral interfaces in Section 4.3.1, then we give a specification of implementation and subtyping relationships in Section 4.3.2.

4.3.1 Behavioral Interface

In this subsection, we introduce the notion of *behavioral interface* for xDSLs. Behavioral interfaces declare the set of domain-specific stimuli that can be send to or received from models conforming to an implementing xDSL.

4.3.1.1 Behavioral Interface Metamodel

Figure 4.5 shows a metamodel formalizing the minimal set of syntactical elements required to define behavioral interfaces and occurrences of the events declared therein and

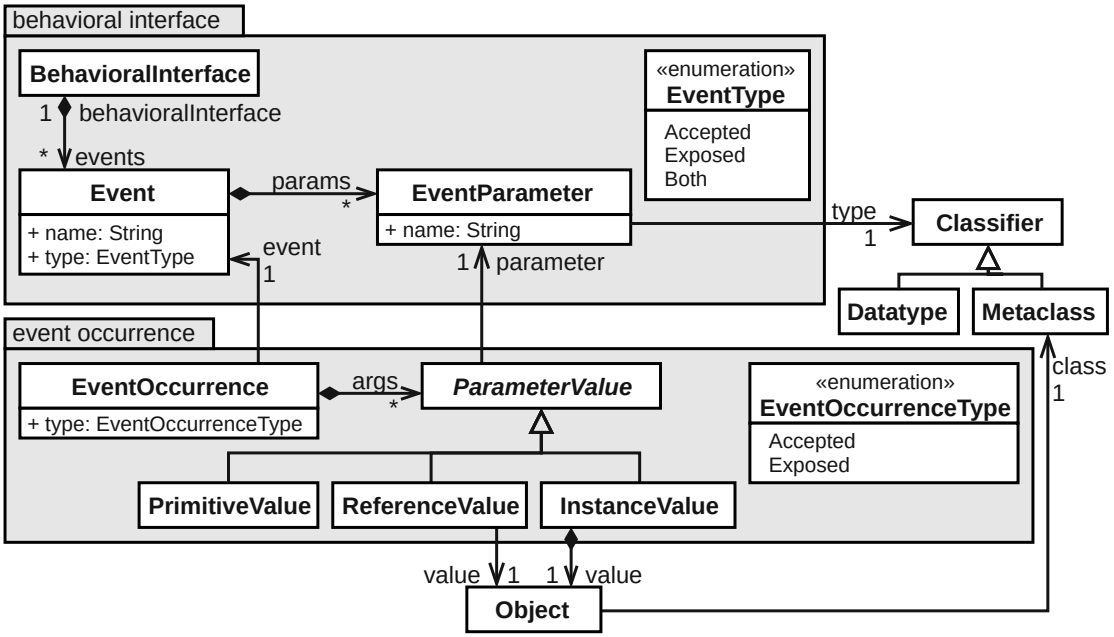


Figure 4.5: Behavioral interface metamodel.

instantiated at runtime. A behavioral interface is composed of Event elements defining the possible interactions with models conforming to xDSLs typed by the interface. Events have a `name` and can either be accepted events, exposed events or both, as indicated by their `type`. Events also have a set of EventParameters that define the data carried by their occurrences. A parameter is identified with a `name` and can either carry primitive values or objects values, as determined by its `type`. Primitive values are typed by a DataType, and object values are typed by a Metaclass. Metaclasses referenced as the type of an event parameter can belong to a specific domain, tying the interface to that domain, or be defined specifically for the behavioral interface (*e.g.,* to carry complex data while keeping the interface self-contained). This allows a behavioral interface to be tied to a specific domain or to be as generic as desired.

To use a behavioral interface, it must be defined as a type for an xDSL, through an implementation relationship between the interface and the xDSL. Alternatively, this can be achieved through a subtyping relationship towards a behavioral interface that is defined as a type for the xDSL. Section 4.3.2 provides more details about implementation and subtyping relationships.

Once this is done, any tool working with xDSLs implementing either any behavioral interface or specific ones can be used with models conforming to the implementing DSL. Such tools send or receive instantiated events from the implemented interfaces under the form of EventOccurrence elements. These elements have an `event` reference to the Event of which they are an occurrence. They also have a `type` attribute indicating whether they are accepted or exposed event occurrences. Additionally, event occurrences contain the

```
BehavioralInterface ArduinoInterface
  accepted run
    parameters [sketch: Sketch]

  accepted button_pressed
    parameters [button: Button]

  accepted button_released
    parameters [button: Button]

  exposed led_level_changed
    parameters [led: Led, level: Integer]
```

Figure 4.6: A behavioral interface for Arduino DSL.

values attributed to each parameter of their Event as ParameterValue elements. According to the type of their corresponding parameter, these parameter values can be PrimitiveValue elements (not detailed in Figure 4.5) or object values. In the case of object values, we make a distinction between references to objects contained elsewhere (ReferenceValues), and objects that are directly contained by the event occurrence (InstanceValues). This allows to pass references to elements of the running model as parameters, as well as objects created for the sole purpose of sending the event occurrence. The referenced model elements are accessed through the read-only structural language interface of the metamodel they conform to, thereby preventing their unauthorized modification. In addition, this allows for event parameters to reference metaclasses that are compatible with several xDSLs, if these references are typed by metaclasses contained in a model type common to these DSLs. In the proposed approach, event occurrences do not need to be contained in another entity. However, a language engineer aiming to provide tooling that revolves around event occurrences can define metamodels (*e.g.,* scenario or trace metamodels) with a composition relationship towards event occurrences.

Definition 3. Let I be a behavioral interface. Occ_I denotes the set of all the event occurrences that can be instantiated from the events defined in I. $Acc_I \subseteq Occ_I$ denotes the subset of all the *accepted* event occurrences, while $Exp_I \subseteq Occ_I$ denotes the subset of all the *exposed* event occurrences.

For instance, in the Arduino DSL, an event signaling the push of a button will carry a reference (thus a ReferenceValue) to the button being pushed, whereas in UML State Machines, an equivalent event would carry an instance of UML event only created to send the event occurrence (thus an InstanceValue), named `"button_pushed"` and itself carrying the identifier of the button being pushed.

4.3.1.2 Examples of Behavioral Interfaces

ArduinoInterface. Figure 4.6 shows a possible behavioral interface for the Arduino DSL, using the textual concrete syntax of the behavioral interface metalanguage. This interface defines three accepted events and one exposed event. As they are accepted events, occurrences of `run`, `button_pressed` and `button_released` can be sent by

```
BehavioralInterface ActivatableInterface
  accepted activate
    parameters [id: String]

  exposed activated
    parameters [id: String]
```

Figure 4.7: The ActivatableInterface behavioral interface.

external tools, triggering specific behavior in executed models. Occurrences of `run` are meant to start the execution of the Sketch element provided for the `sketch` argument. In the case of `button_pressed` and `button_released`, occurrences thereof are meant to change the state of the provided Button element to pressed or released. Conversely, occurrences of the exposed event `led_level_changed` can be emitted and exposed to external tools when specific behaviors are detected in executed models. These occurrences carry two parameters: a Led element and the new value of its `pinValue` attribute.

ActivatableInterface. Figure 4.7 shows a behavioral interface meant for xDSLs whose conforming models contain elements that can be activated, which we refer to as the ActivatableInterface. This language interface can be implemented by xDSLs to extend their definition with the handling of two events relating to the activation of elements: `activate`, which is an accepted event meant to trigger the activation of an element, and `activated`, which is an exposed event notifying that an element has been activated. Both events have an `id` String parameter identifying which element is affected by the event. This makes the interface quite generic and thus usable by various xDSLs, as long as the provided String parameters allows to identify elements of interest.

4.3.1.3 Behavioral Interface Generation

Figure 4.8 shows on the right an excerpt of the most precise behavioral interface (here called ArduinoSignature) that can be defined for the Arduino DSL and, on the left, how it maps to its operational semantics.

For each execution rule of the operational semantics, the behavioral interface contains (*i*) an accepted event triggering calls to the execution rule, (*ii*) an exposed event signaling the start of the execution of the rule, and (*iii*) an exposed event signaling the end of the execution of the rule. Any naming scheme can be used to uniquely name these events. In our case we chose to append or prepend `"called"`, `"returned"` or `"call"` to the name of the execution rule. The parameters of these events are identical to the parameters of their corresponding execution rule.

To streamline the application of the approach to xDSLs, we implemented a generator that systematically derives a behavioral interface to serve as the most precise behavioral type of an xDSL. The generator performs a static analysis of the code of the operational semantics of the DSL to extract all the execution rules it contains. The generator then generates the most precise interface of the DSL based on the signatures of the execution

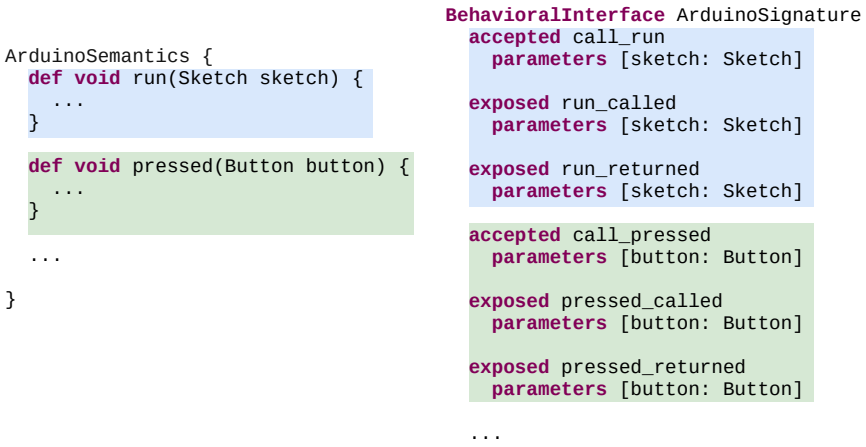


Figure 4.8: Excerpt of behavioral interface (right) derived from the Arduino DSL operational semantics (left).

rules, in accordance with the specification provided above. This generator also provides a corresponding trivial implementation relationship, mapping directly each generated event to its associated execution rule. To complement the provision of such a generator, metalanguage engineers can supply an annotation system or a similar mechanism that allows language engineers to annotate which execution rules will result in accepted and/or exposed events.

4.3.2 Implementation and Subtyping Relationships

In order to use behavioral interfaces as types for xDSLs, it is necessary to define both what is an *implementation relationship* between a behavioral interface and an xDSL, and what is a *subtyping relationship* between two behavioral interfaces. This subsection first lays some preliminary definitions related to operational semantics and events, then specifies both kinds of relationships.

4.3.2.1 Preliminary Definitions

We hereafter introduce the concepts that will be used to specify implementation and subtyping relationships.

Operational Semantics. The proposed approach relies on the translation of accepted event occurrences into actual behavior —*e.g.,* calls to execution rules of the operational semantics— and conversely on the translation of behavior into exposed event occurrences. Essentially, for this approach, interactions with the operational semantics can be considered of two kinds: *call requests* and *call notifications*.

Call requests can be issued to request the execution of a specific execution rule of the operational semantics. Such requests must supply the name of the execution rule to be called, as well as the arguments to be passed when the call is eventually carried out. Additionally, in some cases it may be required to declare that a requested call must be performed in a run-to-completion way, meaning that no other call request should be handled as long as the run-to-completion one has not returned. For example, call requests to the PushButton.press and PushButton.release execution rules of the Arduino DSL should be handled in a run-to-completion way, as calls to these rules model an instantaneous behavior during which nothing else can happen. For this reason, call requests can individually be configured to be handled in a run-to-completion way.

Conversely, call notifications carry a `return` boolean indicating whether a particular execution rule has been or is about to be executed. Call notifications supply the name of the execution rule that has been or is about to be executed, as well as the arguments passed to the execution rule at the moment of the call. Additionally, if the notification informs that an execution rule has been fully executed, it also carries the resulting value of the call, if applicable, as well.

There are numerous ways to define how call requests can be handled by the operational semantics as well as how call notifications are emitted. How this is done depends heavily on the metalanguage used to define the operational semantics of xDSLs. For instance, if the metalanguage being used is a graph transformation language like Henshin [ABJ+10], call requests will likely be handled in between the application of transformation rules, as the model state in the middle of the application of a rule is not consistent. Therefore, we do not restrict our approach to any strategy, but propose one such strategy compatible with our technological space of reference (*i.e.,* where the operational semantics is written in an object-oriented programming language and its execution orchestrated by an execution engine) in Section 4.4.2.

Event Stream. We consider that event occurrences, call requests and call notifications are observed from and inserted into an ordered event stream. This event stream can then be projected on each behavioral interface acting as behavioral type for an xDSL.

Event-Condition-Action Rules. We rely on Event Condition Action (ECA) rules to define implementation and subtyping relationships. ECA rules consist of three parts: an *event part* which specifies which stimuli triggers the rule, a *condition part* which is a predicate that must evaluate to true for the action to be executed, and an *action part* which consists of a behavior to execute when the rule is triggered and its condition satisfied. A wide range of valid strategies exist to define ECA rules, which mostly depend on how the stream of event occurrences is modeled. Therefore, the approach is not restricted to a particular strategy, but one such strategy is proposed in Section 4.4.1.

Event Abstraction Hierarchy. Event abstraction hierarchies consist of layers of abstraction containing complex events defined over the layer below that provide a more detailed or refined view of the event stream.

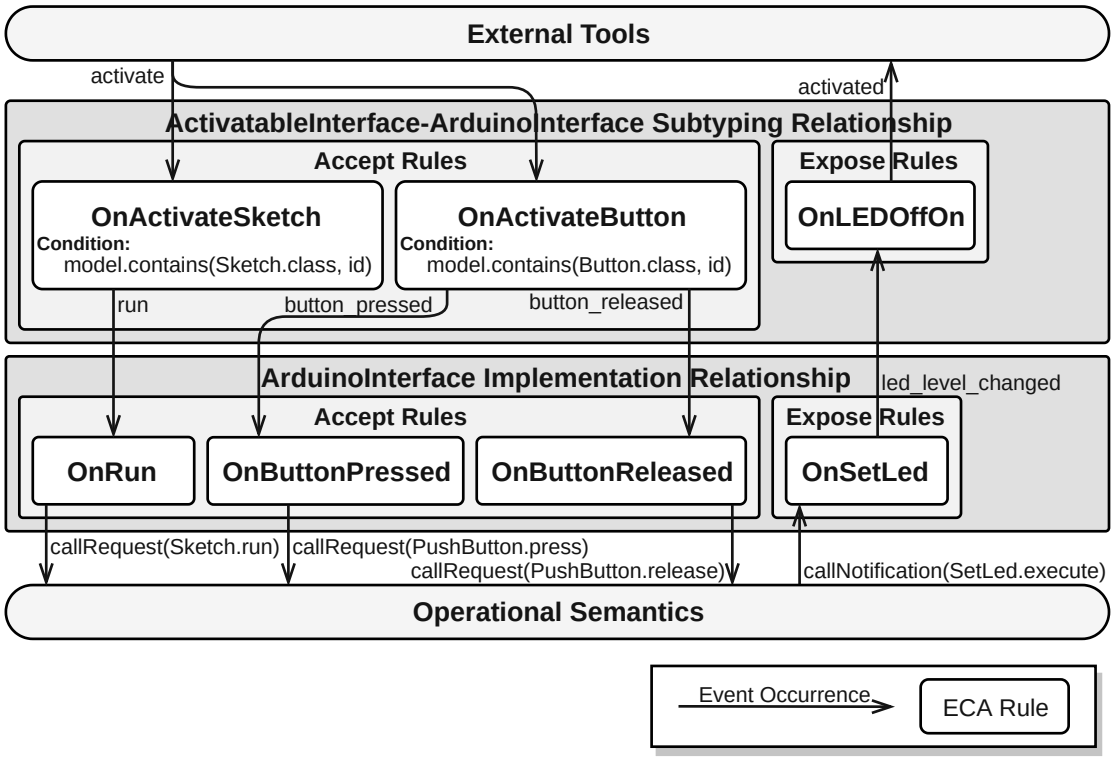


Figure 4.9: Relationships between the Arduino DSL and behavioral interfaces from Figure 4.6 and 4.7.

Pattern. One does not necessarily have control over the definition of the behavioral interfaces involved in a subtyping or implementation relationship. Thus it follows that a one-to-one mapping cannot always be established between the events of two behavioral interfaces (the same applies to events and call requests and notifications). Therefore, to compensate for this potential discrepancy, a means to detect patterns of event occurrences or call notifications is required.

We here consider the definition of temporal pattern matching over a stream of occurrences of events from a behavioral interface, and notifications of calls of execution rules from the operational semantics. Such patterns can be as simple as a single event occurrence, or be more complex like a sequence of several event occurrences in a particular order. Being able to specify and detect temporal patterns in turn enables the definition of ECA rules with a non-trivial event part.

4.3.2.2 Implementation Relationship

A behavioral interface is said to be *implemented* by an xDSL when an *implementation relationship* is defined between the DSL and the interface. Intuitively, an implementation relationship between an xDSL and a behavioral interface guarantees that an observable

behavior from the point of view of the interface can always be defined for every model conforming to this DSL. This means that the implementation relationship translates the internal behavior of models, defined with execution rule calls, into observable behavior, defined with occurrences of events of the implemented interface.

To provide a formal definition of implementation relationships, we rely on labeled transition systems (LTSs) to define the behavior of models and the behavior observable through an interface. We define LTSs as follows:

Definition 4. *Labeled transition system.* An LTS is a tuple $\langle S, L, T \rangle$ where S is a set of states, L a set of labels and T a set of labeled transitions such that $T \subseteq S \times (L \cup \{\tau\}) \times S$.

In addition, we introduce the following notations:

- $p \xrightarrow{\lambda} q$ denotes that there is a transition between p and q which is labeled λ,
- $p \xrightarrow{\lambda_1 \cdot \ldots \cdot \lambda_n} q$ denotes that $p \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} q$,
- $p \overset{\lambda}{\Rightarrow} q$ denotes that there is an arbitrary number of transitions labeled τ and a transition labeled λ such that $p \xrightarrow{\tau} \ldots \xrightarrow{\lambda} \ldots \xrightarrow{\tau} q$,
- $p \overset{\lambda_1 \cdot \ldots \cdot \lambda_n}{\Longrightarrow} q$ denotes that $p \overset{\lambda_1}{\Rightarrow} \ldots \overset{\lambda_n}{\Rightarrow} q$,
- given a set of LTSs LTS, $States(LTS)$ denotes the union set of all the states of the LTSs in LTS,
- given a behavioral interface I, LTS_I denotes the set of all the LTSs that can be defined using a subset of Occ_I as their set of labels.

In our formal definitions, we abstract discrete-event models as LTSs as follows.

- The set of states is defined as the set of possible dynamic states of the model,
- The set of labels is defined as the set of all possible calls that can be performed on the subset of execution rules of the operational semantics exposed by the language engineer (*i.e.,* for which call requests can be accepted, or call notifications sent),
- The set of transitions is defined according to the possible transitions between these states. Transitions that do not involve a call to an execution rule exposed by the language engineer are labeled with τ. In particular, rules whose execution is spread over multiple states and that may be preempted by incoming event occurrences (*i.e.,* rules that are not executed in a run-to-completion fashion) may involve τ-labeled transitions between those execution states, to allow for event-based transitions.

Next, to formally express the behavioral equivalence between the internal behavior of a model and its observable behavior through an interface, we introduce weak and strong parameterized simulation, a variant from weak and strong simulation introduced by Milner in [Mil99], as follows.

Definition 5. *Weak/strong parameterized simulation.* Let L_1, L_2 be sets of labels. Let LTS_1, LTS_2 be the sets of LTSs that can be defined from L_1 and L_2, respectively. Let $\mathcal{S} \subseteq States(LTS_1) \times States(LTS_2)$ be a binary relation. Let $f : L_1 \times States(LTS_2) \rightarrow (\mathbb{N} \rightarrow L_2)$ be a function associating, to a label from L_1 and a state from LTS_2, a sequence of labels from L_2. Then $\mathcal{S}$ is said to be a weak (resp. strong) simulation *parameterized by* f if, whenever $p\mathcal{S}q$, if $p \xrightarrow{\lambda} p'$, then there exists q' such that $q \xRightarrow{f(\lambda,q)} q'$ (resp. $q \xrightarrow{f(\lambda,q)} q'$) and $p'\mathcal{S}q'$. Given two LTSs P and Q, we say that P weakly (resp. strongly) simulates Q through f if there exists a weak (resp. strong) simulation parameterized by f from all states of P to states of Q.

Based on this definition, and more precisely on the definition for weak parameterized simulation, we can the formally define implementation relationships as follows.

Definition 6. *Implementation relationship.* Let $L = \langle AS, \langle DM, ER\rangle\rangle$ be an xDSL and I a behavioral interface. Let DS be the set of all model states conforming to DM, and RC the set of all execution rule calls that can be issued from ER. We say that L *implements* I if there exists a function $Implem : Occ_I \times \mathcal{P}(DS) \rightarrow (\mathbb{N} \rightarrow RC)$ such that, for every model m conforming to AS, there exists an observable behavior $b \in LTS_I$ such that b *weakly simulates* m *through* $Implem$.

In practice, the *Implem* function of an implementation relationship between a behavioral interface and a DSL constitutes a two-layer event abstraction hierarchy, realized through two sets of ECA rules, namely the accept rules and the expose rules. Broadly, accept rules define how event occurrences of the interface are translated into behavior, while expose rules define how behavior results in event occurrences being emitted.

Accept rules define both which event occurrences conforming to the behavioral interface trigger behavior in the executed model, and which parts of the operational semantics of the corresponding xDSL are used for that purpose. Accordingly, the three parts of an accept ECA rule are defined as follows:

- *event*: an accepted event occurrence conforming to the implemented behavioral interface.
- *condition*: a predicate to be applied on the event occurrence and on the model state.
- *action*: specifies, as (possibly run-to-completion) call requests, which (possibly concurrent) sequence of execution rule calls of the DSL semantics must be requested, and with what parameters.

Conversely, expose rules define both which behaviors in the executed model result in event occurrences, and which event occurrences are instantiated thereupon. Each expose ECA rule is structured as follows:

- *event*: a (possibly temporal) pattern to be matched over a stream of call notifications.

– *condition*: a predicate to be applied on the matching set of execution rule calls and on the model state.
– *action*: specifies which exposed event occurrence of the implemented interface must be emitted in response to the detected behavior.

The lower part of Figure 4.9 shows an example of implementation relationship between the ArduinoInterface interface and our Arduino DSL running example. This relationship defines three accept rules and one expose rules. In this case, the ECA rules directly map occurrences of each event to a matching execution rule. For instance, occurrences of `button_pressed` are mapped to a call request for the PushButton.press execution rule and call notifications for the SetLed.execute execution rule are mapped to occurrences of `led_level_changed`. Note that more complex mappings could be included, such as a mapping instantiating two call requests in response to an event occurrence.

4.3.2.3 Subtyping Relationship

A behavioral interface is said to be a *subtype* of another behavioral interface when a *subtyping relationship* is defined between them. Intuitively, a subtyping relationship between two behavioral interfaces guarantees that an observable behavior from the point of view of the supertype interface can always be defined for every observable behavior from the point of view of the subtype interface. This means that the subtyping relationship translates the behavior of models observed through the subtype interface into observable behavior defined with occurrences of events from the supertype interface. We formally define subtyping relationships using strong parameterized simulation as follows.

Definition 7. *Subtyping relationship.* Let I_1, I_2 be two behavioral interfaces. Let $L = \langle AS, \langle DM, ER \rangle\rangle$ be an xDSL implementing I_1, and let DS be the set of all model states conforming to DM. We say that I_1 is a subtype of I_2 if there exists a function $Subtype_{Acc} : Acc_{I_2} \times DS \to (\mathbb{N} \to Acc_{I_1})$, and a function $Subtype_{Exp} : Exp_{I_2} \times DS \to (\mathbb{N} \to Exp_{I_1})$ such that, for every model m conforming to AS with observable behavior $b_1 \in LTS_{I_1}$, there exists an observable behavior $b_2 \in LTS_{I_2}$ such that b_2 *strongly simulates* b_1 *through Subtype*, where *Subtype* is defined as:

$$Subtype\colon Occ_{I_2} \times DS \to (\mathbb{N} \to Occ_{I_1})$$
$$(occ, ms) \mapsto \begin{cases} \text{if } occ \in Acc_{I_2}, & Subtype_{Acc}(occ, ms) \\ \text{if } occ \in Exp_{I_2}, & Subtype_{Exp}(occ, ms) \end{cases}$$

In practice, the *Subtype* function of a subtyping relationship is realized similarly to the *Implem* function of implementation relationships: it constitutes a two-layer event abstraction hierarchy realized with a set of accept ECA rules and a set of expose ECA rules. Subtyping relationships between two behavioral interfaces designate one interface as the *supertype* and the other as the *subtype*. Accept (resp. expose) rules are tasked with translating accepted (resp. exposed) event occurrences of the supertype (resp. subtype) into accepted (resp. exposed) event occurrences of the subtype (resp. supertype). Accept rules are structured as follows:

- *event*: an occurrence of an accepted event from the supertype.
- *condition*: a predicate to be applied on the event occurrence and on the model state.
- *action*: specifies into which sequence of which accepted event occurrences of the subtype the event occurrence is translated.

Expose rules are structured as follows:

- *event*: a (possibly temporal) pattern to be matched over a stream of occurrences of exposed events from the subtype.
- *condition*: a predicate to be applied on the matching set of event occurrences and on the model under execution.
- *action*: specifies into which exposed event occurrence of the supertype the event pattern is translated.

The upper part of Figure 4.9 shows a subtyping relationship between the ActivatableInterface interface as a supertype, and the ArduinoInterface as a subtype. Through this relationship, the language engineer defines the mapping between `activate` and `activated` events and the `run`, `button_pressed`, `button_released` and `led_level_changed` events. In this example relationship, when a button is activated, it means it has been pressed then released. Likewise, a LED is considered as having been activated if it has been switched from off to on. Two accept rules and one expose rule are defined as part of this relationship. The `OnActivateSketch` rule is triggered by occurrences of `activate` and has a condition stating that a Sketch element with a name corresponding to the `id` carried by the event occurrence must exist in the running model. The `OnActivateButton` rule is also triggered by occurrences of `activate` but has a different condition, stating that a Button element with the appropriate name must exist in the running model instead. When triggered, the `OnActivateSketch` rule emits an occurrence of the `run` event, whereas the `OnActivateButton` rule emits two event occurrences: an occurrence of `button_pressed` and an occurrence of `button_released`. The `OnLEDOffOn` rule illustrates that ECA rules can be triggered upon the detection of a pattern of several event occurrences: here, the rule is triggered when a pattern involving several occurrences of the `led_level_changed` is observed. When triggered, this rule directly instantiates an occurrence of `activated`, as it does not have a condition (or rather, its condition always returns `true`).

4.3.2.4 Discussion on Substitutability

Defining implementation and subtyping relationships according to Definitions 6 and 7 guarantees that every model conforming to an implementing DSL has an observable behavior from the point of view of each implemented behavioral interface. This is however not sufficient to guarantee that, if two xDSLs implement the same behavioral interface, *every* model conforming to one DSL can be substituted with at least one model

conforming to the other without any observable difference from the point of view of the implemented interface.

In other words, the fact that an xDSL implements a behavioral interface (either directly or transitively) does not mean that *any* behavior that can be specified with that behavioral interface can be observed from a model conforming to this DSL. An implementation or subtyping relationship achieving this would fulfill a DSL-equivalent of the Liskov substitution principle, as modelers would always be able to use another language implementing the same interface to define a model that can be substituted with a given model.

Proving that an xDSL has such implementation and subtyping relationships defined for a set of behavioral interfaces can be done with a static analysis, provided the xDSL has a formally defined operational semantics.

4.4 Event Management & Metalanguage Integration

Three points remain open in the definition of implementation and subtyping relationships:

1. how to define the ECA rules of a relationship,
2. how relationships deal with (*i*) receiving event occurrences/call notifications, (*ii*) pattern matching of event occurrences and (*iii*) instantiating and forwarding new event occurrences/call requests, namely the event management strategy, and
3. how call requests and notifications are linked to a given operational semantics implementation, namely the metalanguage integration strategy.

In this section, we first propose a possible strategy for event management (in Section 4.4.1), tackling points 1 and 2, and then detail one possible strategy for metalanguage integration (in Section 4.4.2), tackling point 3.

4.4.1 CEP-Based Event Management

Implementation and subtyping relationships, as introduced in Section 4.3.2, require a concrete strategy to define and manage enclosed ECA rules. In this section, we present a strategy based on Complex-Event-Processing (CEP), and more specifically on Esper's Event Processing Language (EPL). First, we mention the salient features of CEP that make it an interesting candidate for event management in our approach. Then we introduce the event manager component acting as an ECA rule engine. Next, we detail how event occurrences are modeled in Esper, and finally we explain the design process of relationships and their ECA rules.

4.4.1.1 Complex Event Processing

The goal of CEP is to identify meaningful events over streams of simpler events with queries on both the data carried by the events and the before and after relationships

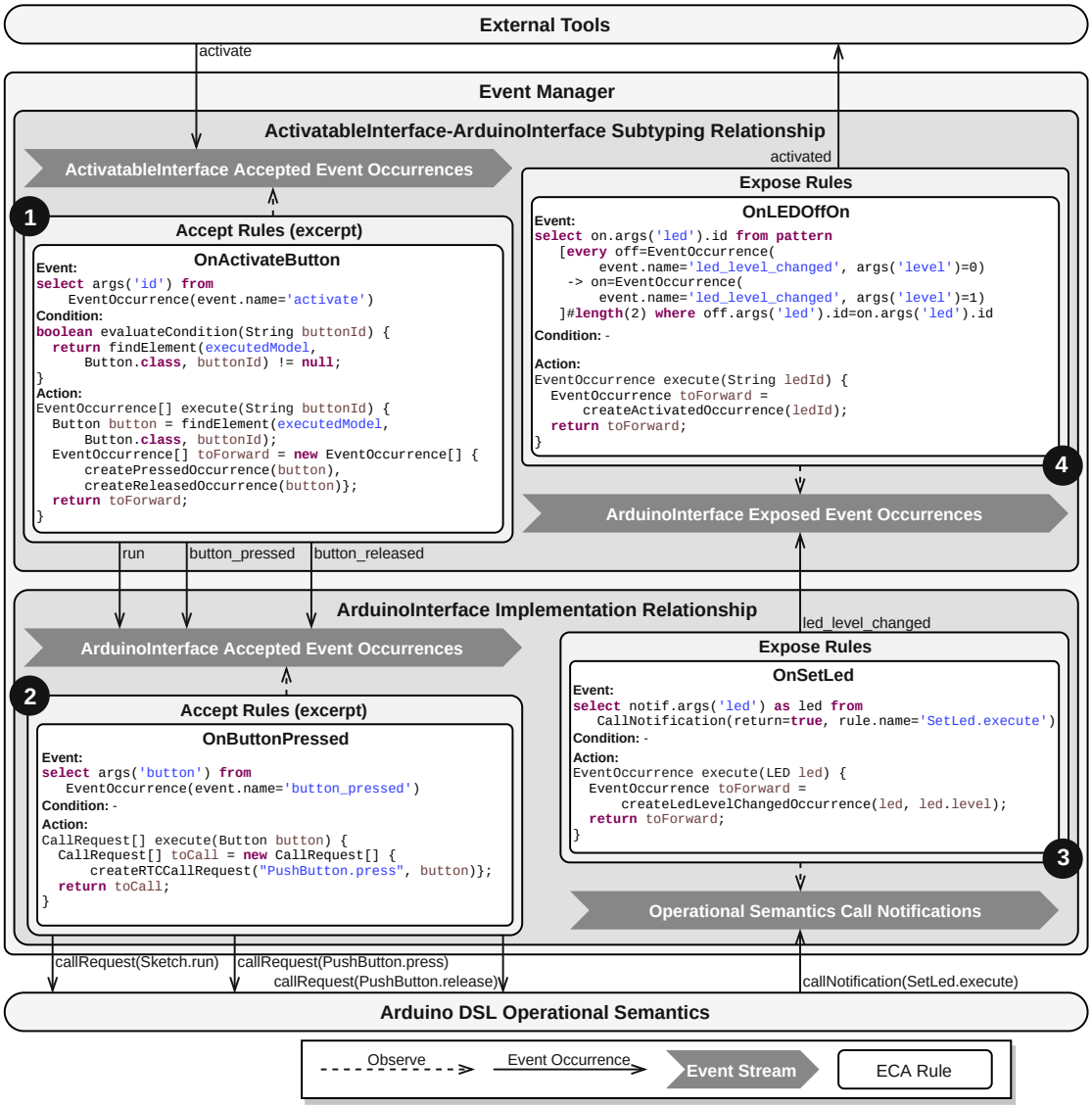


Figure 4.10: Excerpt of the CEP-based architecture applied to the Arduino DSL.

between them. Essentially, CEP systems allow to perform temporal pattern matching over streams of events and produce a new stream of potentially overlapping complex events as a result. In this aspect, CEP is a paradigm that fits particularly well for the definition of event abstraction hierarchies [Luc02], which are central to subtyping relationships between behavioral interfaces.

Esper is an open-source Java-based system for CEP that provides a DSL for event processing called Event Pattern Language (EPL). This DSL allows to formulate queries, called EPL statements, that continuously analyze events within a stream to detect situations of interest and produce a new stream of events containing properties selected

from the matching events. Java objects, called *subscribers*, can then subscribe to this new event stream to be notified each time an event is inserted into the stream.

As we defined the event part of the ECA rules of our relationships as temporal patterns over either a stream of event occurrences or a stream of call notifications, CEP is particularly fitting to the realization of relationships. Moreover, as Esper is Java-based and open-source, it integrates well with our existing model execution framework.

4.4.1.2 Event Manager

To streamline the integration of relationships into the architecture and avoid dependencies between behavioral interfaces, we define a component called the *event manager*. The event manager is implemented as an ECA rules engine configured by the active relationships. For each relationship, two streams are created: one for the environment-to-model direction and one for the model-to-environment direction. According to both the nature of its containing relationship (implementation or subtyping) and its direction (environment-to-model or model-to-environment), a stream contains either event occurrences or call notifications. Streams carrying event occurrences only accepts occurrences of events from the corresponding behavioral interface, based on its direction and on the nature of its containing relationship. The temporal patterns constituting the event part of accept and expose rules are registered to their corresponding stream, as defined by their relationship. The condition and action methods constituting the condition and action parts of the rules of the relationships are then hooked on their corresponding temporal pattern. Figure 4.10 illustrates the event manager, to which an implementation and a subtyping relationship have been registered.

At runtime, the event manager is responsible for dispatching event occurrences between relationships (that is, between their event streams). The event manager dispatches an event occurrence for translation to the event stream of a given relationship based on (*i*) the behavioral interfaces referenced by the relationship, (*ii*) their supertype or subtype role in the relationship (for subtyping relationship only), and (*iii*) the accepted or exposed nature of the event occurrences. Note that, if several registered relationships qualify for a given event occurrence, this occurrence is dispatched to each relationship.

For instance, the subtyping relationship between ActivatableInterface (the supertype) and ArduinoInterface (the subtype) shown on the upper part of Figure 4.10 can receive occurrences of the `activate` and `led_level_changed` events, because `activate` is an accepted event from the supertype interface of the relationship, and `led_level_changed` is an exposed event from the subtype interface of the relationship. However, this relationship cannot receive occurrences of the `activated` event, as this is an exposed event from the supertype interface of the relationship: such event occurrences can be emitted by the relationship but never received, as the relationship does not know how to handle occurrences of this event. For the same reason, this relationship cannot receive occurrences of the `run` event.

Additionally, the event manager is tasked with communicating with the other entities in the system. One such entity is the operational semantics of the xDSL, to which the event manager sends call requests and from which it receives call notifications. Precisely

how this is handled will be discussed in Section 4.4.2. Other possible entities are external tools sending accepted event occurrences to the event manager and/or being notified by the event manager of exposed event occurrences.

Note that, as it has been designed, this event manager is not specific to CEP-based relationships and can accommodate to any technology allowing relationships to offer the following two required services: (*i*) receiving event occurrences and (*ii*) notifying of event occurrences (*e.g.,* using runtime monitors). In fact, an envisioned approach to define relationships is to propose a dedicated, declarative event mapping language letting the language engineer define when and on what condition an event or sequence thereof should be mapped to another event or sequence thereof.

4.4.1.3 Modeling Event Occurrences in Esper

To use Esper we need to map our event occurrences to event representations that can be processed by Esper. A range of possibilities are available, from Plain Java Objects (POJOs) to Maps to XML documents. We opted for modeling our events as POJOs, as we do not require the flexibility of Maps, and our implementation is exclusively Java-based, making XML both cumbersome and unnecessary. More specifically, we defined a wrapper class for EventOccurrence objects. This wrapper class declares two methods, that are considered as event properties by the Esper runtime. The first method is the getEvent method, which returns the `event` of the occurrence. The second method is the getArgs method, that takes an event parameter name as parameter and returns the value associated to that parameter. This allows Esper to access the different arguments of an event occurrence as a mapped property, by supplying the parameter name of the argument. For instance, the expression `args('someName')` returns the value provided for the event parameter named `'someName'`.

As in our proposed strategy, call notifications are inserted into the event stream and manipulated by the Esper runtime, we also need an Esper representation for them. Since call notifications are issued by the integration facade, which in our case is Java-based, the simplest solution for the proposed architecture is to model these call notifications as POJOs, as we do for event occurrences. Such POJOs point to the execution rule at the origin of the call notification, to a map associating the values supplied for each parameter of the execution rule in that particular call, and for notifications of completed calls, to the value returned by the call.

4.4.1.4 Relationship Design

With event occurrences and call notification made Esper-compatible, we can now look into the design process of implementation and subtyping relationships and their ECA rules, based on Esper and Java. We will then present a concrete example of the application of this process to our Arduino DSL running example.

Design Process. ECA rules are defined in the following manner. The event part of ECA rules is defined using EPL statements querying the stream corresponding to the

nature of the rule (*i.e.,* accept or expose). This allows to leverage the power of CEP to capture complex, potentially overlapping patterns of event occurrences. The condition and the action parts of a rule are written as Java methods to be called by the event manager when a complex event is detected by the EPL statement defined as the event part. The condition method takes complex events detected by the EPL statement as parameter, and returns a boolean value indicating whether the action method should be called or not. To be able to enforce domain-specific constraints, the condition method has access to the running model in addition to the triggering complex event to compute its result value. Conversely, the action method also takes as parameter the complex event that was detected by the EPL statement. The action method of accept rules returns either an array of event occurrences (for subtyping relationships) or an array of call requests (for implementation relationships), while the action method of expose rules always returns a single event occurrence. Access to the running model allows the action method to configure newly instantiated event occurrences (*e.g.,* supplying event occurrences with parameters from the model).

Concrete Example. Figure 4.10 illustrates this design strategy by showing a more in-depth view of Figure 4.9, which provides an overview of implementation and subtyping relationships between ActivatableInterface and ArduinoInterface. First, it highlights the fact that each relationship holds two event streams: a stream associated to accept ECA rules (next to labels **1** and **2**), and another associated to expose ECA rules (next to labels **3** and **4**).

Then, on the upper-left part of the figure (labeled **1**), the OnActivateButton accept rule of the subtyping relationship between ActivatableInterface and ArduinoInterface is detailed. The event stream observed by this rule contains ActivatableInterface accepted event occurrences. The event part of the OnActivateButton rule is an EPL statement that notifies its subscribers (*i.e.,* the registered rules) whenever an occurrence of an event named `activate` is inserted on the event stream. When this happens, the subscribers receive a notification that carries the `id` parameter value, selected by the EPL statement through the `args('id')` expression. In the example, there are two subscribers, one of which (OnActivateSketch) is not shown. The other subscriber is the OnActivateButton rule. When notified, the evaluateCondition method, whose implementation is required of subscribers, is called. This method checks that the condition of the rule is satisfied. In the example, the implementation of this condition method performs a query on the running model, using the value provided by the complex event pattern of the event part of the rule. This is achieved using a utility method findElement which finds an element of the provided class with the provided name (here `buttonId`) in the provided model (here the running model). Then, if the condition is satisfied, the subscriber performs the action of the rule by calling its execute method, which translates the triggering event occurrence into two new event occurrences. This is effectively done by instantiating the new occurrences (using dedicated utility methods in our example) and returning them in an array to be inserted in the correct event stream (in this case, the stream of ArduinoInterface event occurrences).

On the lower-left part of the figure (labeled **2**), the design of the accept rules of the ArduinoInterface implementation relationship is detailed. It is very similar to that of the subtyping relationship, the event stream containing ArduinoInterface accepted event occurrences instead of ActivatableInterface ones. The accept rules observing this event stream instantiate and return call requests for specific execution rules. The OnButtonPressed rule shown on the figure detects occurrences of the `button_pressed` event and converts them directly (as no condition is specified) into call requests for the PushButton.press execution rule of the operational semantics.

Then, on the lower-right part of the figure (labeled **3**) is detailed the design of the expose rule of the implementation relationship. The event stream associated to this rule contains the call notifications issued by the operational semantics. The OnSetLed rule detects call notifications for the SetLed.execute execution rule on the event stream, and converts them into occurrences of the `led_level_changed` event, to which it supplies the referred `led` and its new `level`.

Finally, on the upper-right part of the figure (labeled **4**) is detailed the OnLED-OffOn expose rule of the subtyping relationship. This rule observes an event stream containing ArduinoInterface exposed event occurrences. The EPL statement constituting the event part of the rule specifies that it is triggered whenever a succession of two `led_level_changed` event occurrences with alternating `level` parameter values but identical `led` parameter values is observed in a sliding window of 2 events. The action part of the rule translates the triggering complex event into an occurrence of the `activated` event with the `id` of the LED as a parameter value.

4.4.2 Metalanguage Integration

In addition to providing a unified way to define the accepted and exposed events for any xDSL, our approach aims to be agnostic of the metalanguage used to define the operational semantics of an xDSL. This means that the behavioral interface language and the design of the event manager and relationships must work for any xDSL, regardless of the metalanguage used to defined its operational semantics.

To achieve this, an integration facade for the event manager must be defined. This facade is tasked with translating call requests into actual behavior, and behavior into call notifications, thereby bridging the gap between the event manager (and the implementation relationships therein) and the operational semantics.

In this section, we propose such an integration facade to enable the approach for xDSLs whose operational semantics is defined using an object-oriented metalanguage such as Java and orchestrated by an execution engine. Note that the proposed integration facade is intended for sequential model execution. Adapting the approach to concurrent model execution only requires to define an appropriate integration facade. First, we present what must be provided by this execution engine, which is considered as a prerequisite for the proposed facade. Next, we detail the inner workings of the integration facade. Finally, we show how this facade is interfaced with the aforementioned execution engine.

4.4.2.1 Execution Engine

The proposed approach considers that a pre-existing *execution engine* applies the operational semantics of the considered xDSL on the running model. Such an execution engine must be able to notify external components when it starts or stops, and when it applies execution rules that alter the model state. More precisely, the engine only sends notifications for execution rules annotated as a *stepping rule*, which are executions rule producing an observable execution step when applied. Regarding the Arduino DSL presented in Section 4.1, only stepping rules are presented.

The state of the model is considered observable and alterable at the time notifications are made and handled; hence the possible observable states reached during an execution are heavily dependent on the granularity of the declared stepping rules in a semantics. This notification mechanism can be used to attach interactive debuggers [BLC+18] and trace constructors [BMCB17] to the execution. We explain later how we further leverage it to enable exposed events and run-to-completion call requests. The design of an execution engine is described in more detail in our previous work [BLC+18, BDV+16], and can be summarized as the following operations:

- start does the ensuing actions:
  - load the considered xDSL;
  - load the model to be executed;
  - register the execution observers;
  - prepare the initial model state;
  - set the `running` attribute of the engine to `true`;
  - notify registered observers that it is starting.
- stop sets the `running` attribute to `false`, and notifies execution observers that the engine is stopping.
- callExecutionRule starts the application of a specific execution rule of the operational semantics. If it is a stepping rule, the engine notifies observers at the beginning and at the end of the execution of the rule. Note that depending on the metalanguage, an execution rule may trigger the nested execution of other stepping rules, in which case observers are also notified when the nested execution of these stepping rules begins or ends. For instance, in the Arduino model shown in Figure 4.2, calls to SetLed.execute will be nested within calls to If.execute, which will in turn be nested within calls to Sketch.run. Note that no distinction is made between the notifications from nested and non-nested rule calls.
- registerObserver registers a component as an *observer* that gets notified when the execution of a stepping rule begins or ends, and when the engine starts or stops. When an observer gets registered, an associated *priority policy* needs to be supplied as well. Such a policy provides, for each kind of notification, the priority at which the observer must be notified. This operation is called by the execution engine during the initialization phase, to register a predefined set of execution observer, retrieved from a configuration file for instance, but it can also be called at any time.

The specification of this component is by design as generic as possible to be able to cover a wide range of metalanguages. As such, it provides an abstraction over the multiple execution engines dedicated to the various metalanguages available in the GEMOC Studio (see Section 7.1). The implementation of this component is however heavily dependent on the metalanguage used to implement the operational semantics, especially regarding the procedure to dynamically call an arbitrary execution rule (*e.g.,* using `java.lang.reflect.Method.invoke` if the semantics is implemented in Java).

Using these operations, a user (*e.g.,* a modeler, a tool) is able to execute a model by starting the engine, then demanding the execution of one or several execution rules of the semantics (*e.g.,* a *run* method responsible for the complete execution). In the following subsections, we explain how the integration facade can also use these operations for managing call requests and notifications.

4.4.2.2 Overview of the Metalanguage Integration Facade

To bridge the gap between implementation relationships and the execution engine, we define an integration facade concentrating on the following two activities: (*i*) waiting for execution rule call requests from implementation relationships and performing the requested calls, and (*ii*) issue execution rule call notifications to implementation relationships.

To be able to perform these activities, the integration facade has two requirements that need to be fulfilled. First, it needs a mechanism to wait for execution rule call requests to arrive. To that effect, our approach relies on a *blocking queue* to store the call requests received from implementation relationships. Call requests can be retrieved from the queue using the poll and the take operations, which behave differently when the queue is empty: take suspends the execution and waits for an element to be available, while poll simply returns `null`. Second, the integration facade needs to be able to call execution rules defined as part of the operational semantics of an xDSL. This task is delegated to the execution engine and its callExecutionRule operation.

With these requirements fulfilled, an execution with the proposed integration facade unfolds as follows:

- The integration facade is notified of the start of the execution by the execution engine. It enters its execution rule call scheduling loop: the execution is repeatedly suspended when the call request queue is empty, and resumed when call requests are queued.
- Implementation relationships send call requests to the integration facade, which are added to the call request queue.
- The engine informs the integration facade when it is safe to process the queued call requests, *i.e.,* when starting or ending stepping rule calls. In such cases, the integration facade first checks whether a run-to-completion call request is currently being executed. If that is the case, the call request queue is left untouched. Otherwise, the queued call requests are sequentially delegated to the execution engine.

Algorithm 4.1: startListening

```
Input :
  engine      :  the execution engine,
  callReqQueue :  the call request queue
1 callRequest ← callReqQueue.take();
2 while engine.running ∧ callRequest ≠ Stop do
3 |   processCallRequest(callRequest)
4 |   callRequest ← callReqQueue.take()
5 end
```

– The integration facade is notified that a stepping rule call is about to start or has ended, and forwards this notification to implementation relationships.

4.4.2.3 Metalanguage Integration Facade Operations

We hereby present how the integration facade achieves these different tasks through a set of operations.

startListening and stopListening. These internal operations are used to start and stop the call request handling loop. Algorithm 4.1 shows startListening. As long as the execution engine is `running`, the first call request of the call request queue (lines 1–2 and 4 of Algorithm 4.1) is retrieved. When the take operation is called on the queue, the execution is suspended if the queue is empty —which only happens if no execution rule is currently executing— and resumes as soon as a request is added. Finally, the call request is processed using the processCallRequest operation (line 3 of Algorithm 4.1). The stopListening operation consists of inserting an instance of a special Stop call request into the call request queue (mechanism know as a *poison pill* [GPL+06]), thereby stopping the call request handling loop.

queueCallRequest. This operation is called by the event manager to insert a request to call the provided execution rule with the provided arguments into the call request queue. Note that, at the start of the execution, no actual execution takes place until a first call request is queued. For instance, in the case of the Arduino DSL, the execution only starts once a `run` event occurrence is received: this event occurrence enqueues, through a call to the queueCallRequest operation, a request for a call to Sketch.run on the `sketch` parameter of that event occurrence. This call request is then processed, which starts the execution.

manageCallRequests. This internal operation is similar to startListening, except that it does not suspend the execution when the queue of call requests is empty. It is called when the integration facade is notified that the running model is in a consistent state and thus that pending call requests can be safely handled. As explained previously, this

Algorithm 4.2: manageCallRequests

Input:
engine : the execution engine,
callNotification : the notification,
callReqQueue : the call request queue,
callStack : the call stack

```
1 call ← callStack.peek();
2 if ¬call.runToCompletion ∧ call.rule ≠ callNotification.rule then
3 |   callRequest ← callRequestQueue.poll();
4 |   while callRequest ≠ null ∧ callRequest ≠ Stop∧ engine.running do
5 |   |   processCallRequest(callRequest);
6 |   |   callrequest ← callRequestQueue.poll();
7 |   end
8 end
```

Algorithm 4.3: processCallRequest

Input:
engine : the execution engine,
callRequest : the call request to process,
callStack : the call stack

```
1 callStack.push(callRequest);
2 ruleToCall ← callRequest.rule;
3 engine.callExecutionRule(ruleToCall, callRequest.args);
4 callStack.pop()
```

is the case before and after the execution of stepping rules. Algorithm 4.2 shows the behavior of this operation. When it is called, the integration facade first checks that the currently executed call request did not ask for run-to-completion behavior. For this, the call request on top of the stack is inspected (line 1) and two conditions are checked: if it should not be treated as run-to-completion, and if its associated execution rule is different from the stepping rule that triggered the notification (line 2). The first condition prevents the processing of a call request while a run-to-completion call request is being handled. The second condition prevents the processing of additional call requests before the processing of the current one gets to start, which would otherwise happen when the rule associated to the current call request is a stepping rule. If both conditions allow it, the non-blocking poll operation is used to iterate over all call requests in the queue and process them using the processCallRequest operation (lines 3–7), exiting the loop if the engine stops or if the Stop call request is encountered. Otherwise, the call request queue is left untouched, to be processed at a later time, as the operation returns immediately.

processCallRequest. This internal operation, detailed in Algorithm 4.3, is used to process a single execution rule call request. First, the call request is pushed on a *call stack* (line 1). Then, the execution rule to call is retrieved from the call request (line 2), and the call is delegated to the execution engine (line 3). Once this call returns, the call request is popped from the call stack (line 4). This call stack keeps track of the call requests that are currently being handled and is used to enforce the potential run-to-completion nature of call requests by preventing the handling of other call requests while a run-to-completion one is being executed.

4.4.2.4 Integration with the Execution Engine

During its initialization phase, the execution engine instantiates and registers the integration facade as an observer from a configuration file. In the following, we detail how the integration facade reacts to the different notifications sent by the execution engine, combining the presented operations to achieve proper event handling.

- *notifyStart*: the call request handling loop is started, using the startListening operation.
- *beforeStep*: the manageCallRequests operation is called to process the call request queue, given that the call request currently under execution (if any) is not a run-to-completion call request.
- *afterStep*: call notifications are forwarded to implementation relationships, which decide if they should result in an exposed event occurrence. The facade then behaves as for *beforeStep* notifications.
- *notifyStop*: the stopListening operation is called to halt the call request handling loop.

In the event that all execution rule calls issued from the startListening operation terminate without a *notifyStop* notification being received, the call request handling loop suspends the execution, waiting for either a Stop request or a call request to be queued and thus instantly processed. Note that, when the integration facade is registered as an observer of the execution, an accompanying priority policy is supplied, specifying that it receives *notifyStart* and *afterStep* notifications last, but receives *beforeStep* notifications first. This allows the facade to work with other potential execution observers. For instance, a trace constructor needs to receive *beforeStep* notifications after the integration facade: otherwise, it would record the start of an execution step when in fact another step could be triggered, given there is a pending call request in the queue of the integration facade.

4.5 Evaluation

In this section, we first evaluate whether the proposed approach fulfills each of its requirements, which are listed in 4.1, then we conclude by summarizing and discussing the results, as well as the threats to validity.

```
BehavioralInterface StateMachineInterface
  accepted run
    parameters [stateMachine: StateMachine]

  accepted signal_received
    parameters [signal: SignalOccurrence]

  accepted call_performed
    parameters [call: OperationCall]

  exposed signal_sent
    parameters [signal: SignalOccurrence]
```

Figure 4.11: Behavioral interface for UML State Machines.

4.5.1 Interface Definition and Implementation (Req. 1)

To evaluate how well the proposed approach fulfills **Req. 1**, we apply the approach on two existing xDSLs to enable interaction with their conforming models. In a first time, we apply the approach on the Arduino DSL presented in Section 5.1, a very specific DSL. In a second time, we apply the approach to a subset of UML State Machines in conformance with the Precise Semantics of UML State Machines (PSSM) specification [Obj19], which is a general and standardized modeling language. We then report on the process.

Executable DSL I: Arduino. The first xDSL on which we apply the approach is the Arduino DSL presented in Section 5.1. The behavioral interface directly implemented by the Arduino DSL has been introduced in Figure 4.6 and contains three accepted events (`run`, `button_pressed` and `button_released`) and one exposed event (`led_level_changed`). The implementation relationship defined between this behavioral interface and the operational semantics of the Arduino DSL is straightforward:

- `run` occurrences are translated into call requests to the Sketch.run execution rule,
- `button_pressed` and `button_released` occurrences are translated into call requests to the Button.press and Button.release execution rules, and
- calls to the SetLed.execute execution rule are translated into `led_level_changed` occurrences, with the new level being directly queried from the model.

In total, the implementation relationship itself required, for each ECA rule, around 5 to 6 lines of Java code for the method bodies, while we were able to define a library specific to Esper-based implementation relationships that can be reused for any implementation relationship. The definition of this library required 94 lines of Java code.

Executable DSL II: UML State Machines. The second xDSL on which we applied the approach is a subset of UML State Machines in conformance with the Precise Semantics of UML State Machines (PSSM) specification [Obj19]. Since we focused on reproducing the event-related behavior of UML State Machines with our approach, we implemented a relevant subset of the language defined as follows:

- The implementation supports initial, final, entry point, exit point, fork, join and terminate pseudo-states. History, choice and junction pseudo-states are not supported as they take no part in the event-handling behavior of UML State Machines.
- Although PSSM is an extension of fUML [Obj13b], which gives semantics for UML Activity Diagrams, our implementation of PSSM only covers UML State Machines.
- State machine redefinition is not supported since this is not related to the event handling logic.

Among the execution rules of the operational semantics, 4 rules stand out: the StateMachine.run starts the execution of the model, StateMachine.signalReceived notifies a StateMachine that it received a SignalOccurrence, StateMachine.callPerformed notifies a StateMachine that call was performed, and Behavior.execute launches the execution of a Behavior.

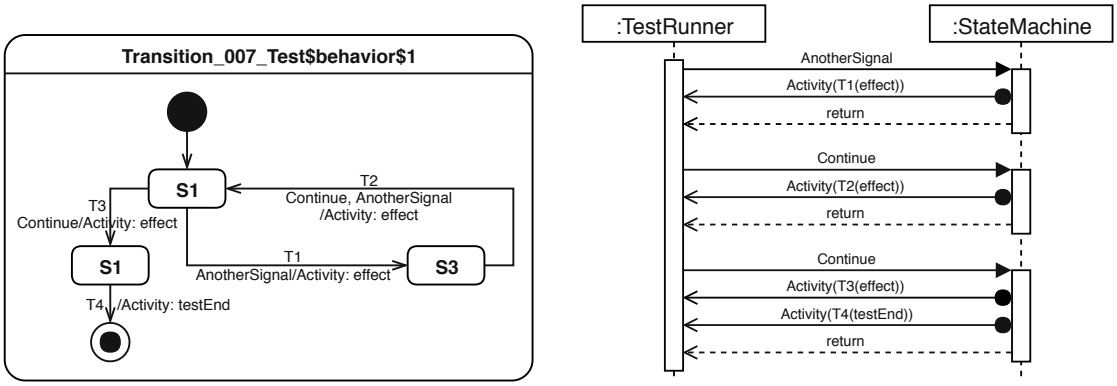
Figure 4.11 shows the StateMachineInterface we defined for UML State Machines. This interface contains three accepted events that are described thereafter. The `run` event triggers the initialization required to start the execution of the state machine. The `signal_received` event takes a signal occurrence as parameter and triggers run-to-completion steps. As signals potentially contain parameters, signal occurrences can provide values for these parameters. The `call_performed` event takes an operation call as parameter and also triggers run-to-completion steps. The interface also contains one exposed event: `signal_sent`, which takes a signal as parameter. Note that this event normally occurs in the activity diagrams used to define the behavior of states and transitions of UML State Machines, not in the state machines themselves. However, as our implementation does not include Activity Diagrams, instead using stubs thereof, we added the `signal_sent` event to the StateMachineInterface.

As for the Arduino DSL, the implementation relationship defined between the StateMachineInterface and UML State Machines is straightforward. First, `run`, `signal_received` and `call_performed` occurrences are translated into call requests for the run, signalReceived and callPerformed execution rules, with a one-to-one mapping between event occurrence arguments and execution rule arguments. Second, call notifications for the Behavior.execute execution rule are translated into `signal_sent` occurrences.

In total, the implementation relationship itself required, for each ECA rule, around 5 to 8 lines of Java code for the method bodies, and we were able to reuse the same library for Esper-based implementation relationships that we defined when applying the approach to the Arduino DSL.

Fulfilling Req. 1. We successfully applied the approach on two xDSLs, one very specific and the other more general. In the process, we defined a library that language engineers can reuse to define their Esper-based implementation relationships. This allowed to keep the amount of lines of code required to implement each ECA rules very low, at around 5 to 8 lines of Java code.

This shows that the proposed metalanguage is expressive enough to define in a unified way the possible interactions with models conforming to these two DSLs. The soundness



(a) Test model from the PSSM test suite.

(b) Sequence diagram of test case execution from the PSSM test suite.

Figure 4.12

of the enabled model interactions is up to the language engineer, as it depends on the execution semantics of the DSL. The language engineer has full control over which behavior can be triggered by event occurrences, and the conditions of the ECA rules allow the language engineer to perform extensive checks before accepting or emitting event occurrences. Therefore, the approach fulfills **Req. 1** for the two considered DSLs.

4.5.2 Realizing Reflective Tools (Req. 2)

We evaluate **Req. 2** by demonstrating how the proposed approach provides genericity through reflection. In more details, we demonstrate how the reflection capabilities provided by our metalanguage for behavioral interfaces enables the development of tools compatible with any xDSL implementing a behavioral interface. Consequently, language engineers applying the approach to define a behavioral interface and an implementation relationship for their xDSLs are able to provide some degree of interactive tool support for free, as their DSLs directly benefits from reflective tool support. We first demonstrate how a test runner able to run test suites for any xDSL can be defined. We then show that the approach enables the definition of a GUI to configure and send accepted event occurrences, and receive exposed event occurrences in accordance to the definition of their events. Combined together, these two tools allow for practical definition and execution of test cases, for instance for non-regression testing. Indeed, the GUI can be used to configure accepted event occurrences, send them and store both these occurrences and the ones received in return under the form of a test scenario that can then be run by the test runner.

Reflective Tool I: Test Runner. As a first reflective tool, we implemented a test runner which is able to process a previously defined test suite to drive the execution of a model under test and check an oracle.

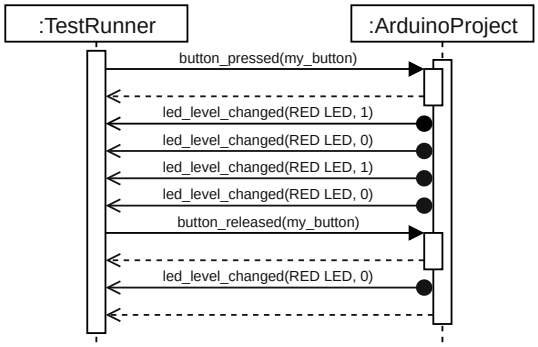


Figure 4.13: Sequence diagram of a test case for the Arduino model shown in Figure 4.2.

Test cases defined in a test suite point to an xDSL definition, as well as a model under test conforming to that DSL. Each test case also contains both a test scenario as a sequence of event occurrences to send to the model, and an oracle as event occurrences that must be received, interleaved with the test scenario. The test runner, implemented as a launch configuration for the GEMOC Studio, reads provided test suites and iterates over the test cases they contain. For each test case, the test runner starts a new execution using the operations detailed in Section 4.4.2.1. Then, the test runner alternates sending event occurrences from the test scenario of the test case, and waiting for event occurrences from the oracle of the test case. The implementation of this tool is possible due to the unified representation of events and their occurrences provided by the approach, as Event and EventOccurrence elements. Indeed, this both enables to define test suites that contain event occurrences from any behavioral interface, and allows the test runner to send and receive event occurrences while staying agnostic to their behavioral interface.

Using this tool, we were able to check the conformance of our implementation of a subset of UML State Machines with the Precise Semantics for UML State Machines (PSSM). We retrieved the available test suite designed for Papyrus and, using a model transformation, converted it into a test suite model compatible with our test runner. Figure 4.12a shows a test model taken from a test case of the test suite. The aim of this test case is to verify that a transition containing multiple triggers (here `T2`) can be fired if at least one of these triggers matches a received signal. Figure 4.12b shows a sequence diagram illustrating how, for the considered test case, the test runner interacts with the test model according to the test scenario and oracle.

We also used the test runner to execute test cases on Arduino models. Figure 4.13 illustrates such a test case for the `blinking` sketch shown in Figure 4.2. In this scenario, the button is pressed, causing the LED to start blinking. Once the LED has blinked twice, the button is released and the test runner waits for a last exposed event occurrence indicating that the led has effectively stopped blinking, and the test case ends.

Reflective Tool II: Event Injection GUI. As a second reflective tool we implemented a reflective event injection GUI for the GEMOC Studio that leverages the active

behavioral interfaces to (*i*) allow the user to create and send accepted event occurrences to the running model, and (*ii*) listen to exposed event occurrences and display them in a log. In more details, the tool features a list of all implemented and supertype interfaces. For each behavioral interface selected in this list, the tool provides an event occurrence configurator per accepted event defined in the interfaces. By reflectively analyzing the defined parameters for each accepted event, the GUI is able to provide well-suited controls to configure an occurrence of these events, such as a text field letting users enter the value of their choosing for parameters whose type is a string (*e.g.,* the `id` parameter of the `activate` event). Alternatively, the configurator for `button_pressed` event occurrences provides a list of all model elements whose type matches the parameter type (PushButton in this case), as well as a browse button that lets users select a predefined model element in an arbitrary resource located in the workspace. Finally, the GUI provides a log of exposed event occurrences listing all the received event occurrences.

As we implemented our approach and tools within the GEMOC Studio, we are able to use the reflective event injection GUI in conjugation with the generic debugger already provided by the GEMOC Studio [BLC+18]. Using this extended debugger, we are able to pause the execution, queue event occurrences, use stepping operators (forward and backward), define breakpoints, and resume a paused execution to evaluate the impact of queued event occurrences on this execution. This can be used on any model conforming to any DSL developed with any of the metalanguages provided by the language workbench for which an integration facade is defined.

Fulfilling Req. 2. No tool-specific line of code is required to interact with running models, indicating that the event manager component provides a sufficiently expressive API for both tools. In addition, by design, the event manager guarantees that event handling does not result in undefined behavior, as it forbids simultaneous calls to execution rules. The condition part of ECA rules also guarantees that execution rules are only called in execution states allowed by the language engineer.

Therefore, by implementing these two tools and showing how they can be used with both UML State Machines and Arduino DSL, we showed that reflective tools can be built that leverage behavioral interfaces to both discover how to interact with a running model and do so in a sound and unified way. This means that the approach fulfills **Req. 2** for these two tools.

4.5.3 Interface Subtyping (Req. 3)

We show how the approach fulfills **Req. 3** by combining implementation and subtyping relationships defined over an xDSL and its behavioral interfaces to define an event abstraction hierarchy. This to define interactive tools tailored for a given behavioral interface (*i.e.,* a given set of top-level events), and yet that can be reused across all xDSLs implementing this interface, either directly or transitively. In more details, we define a subtyping relationship between the ActivatableInterface interface shown in Figure 4.7 and both the ArduinoInterface and the StateMachineInterface. We then discuss how this

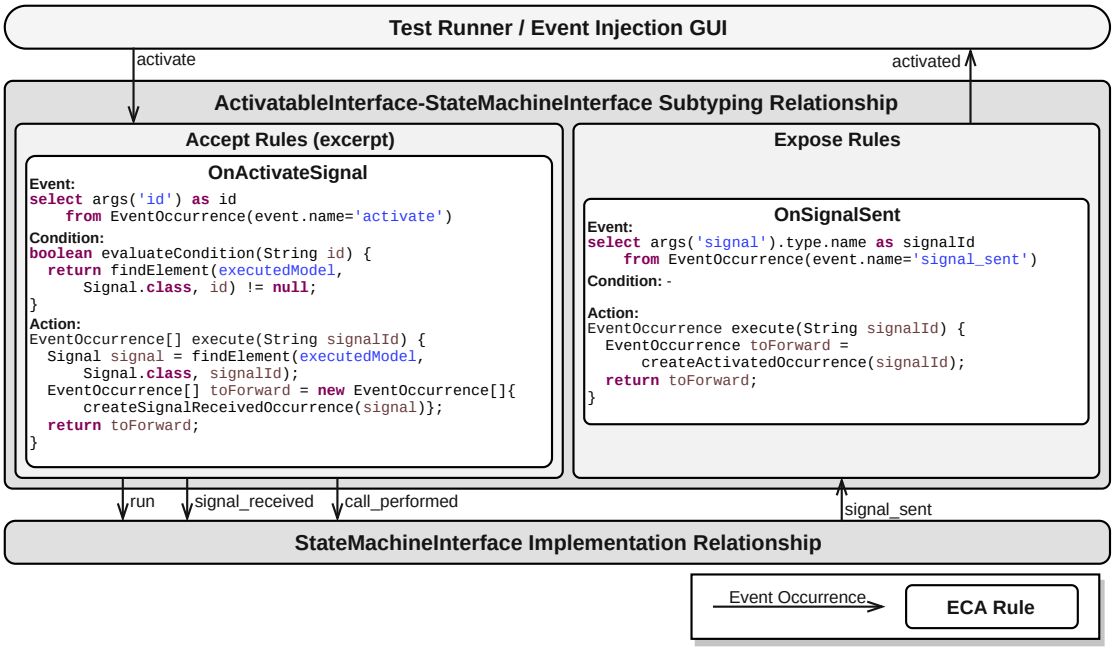


Figure 4.14: Subtyping relationship between ActivatableInterface and StateMachineInterface.

enables the interchangeability of the two considered xDSLs, and what the reaped benefits are.

Subtyping with Arduino DSL. As the subtyping relationship between ActivatableInterface and ArduinoInterface has already been presented in Figures 4.9 and 4.10, we will summarize its content thereafter.

This subtyping relationship features two accept rules translating `activate` events into either a `run` event occurrence, or a sequence of two event occurrences: a `button_pressed` occurrence followed by a `button_relea-sed` occurrence. This depends on whether the `id` parameter value of the event occurrence refers to a Sketch or to a PushButton. The relationship also features one expose rule translating `led_on` and `led_off` occurrences into `activated` occurrences. This is a more complex ECA rule as its event part consists of a pattern that matches sequences of `led_off` occurrences followed by `led_on` occurrences on a specific time frame. Only when a match is found can the involved event occurrences be translated into a `activated` occurrence.

As for implementation relationships, we were able to define a library specific to Esper-based subtyping relationships that we reused for all subtyping relationships we defined. By leveraging this library, the ECA rules of the subtyping relationship required around 3 to 5 lines of Java code per method body.

Subtyping with UML State Machines. Figure 4.14 details the content of the OnActivateSignal and OnSignalSent ECA rules of the subtyping relationship between ActivatableInterface (as a supertype) and StateMachineInterface (as a subtype).

The left part of the figure highlights the fact that `activate` occurrences (from ActivatableInterface) are translated into `run`, `signal_received` and `call_performed` occurrences (from StateMachineInterface). The OnActivateSignal rule is detailed. Upon detecting an `activate` occurrence it first checks that a Signal element with a name identical to the value supplied for the `id` parameter of the `activate` occurrence exists in the executed model. If that is the case, the rule translates the original event occurrence into a `signal_received` occurrence from StateMachineInterface. This new occurrence is configured to carry a newly instantiated occurrence of the proper signal.

The right part of the figure highlights the fact that `signal_sent` occurrences (from StateMachineInterface) are translated into `activated` occurrences (from ActivatableInterface) through the OnSignalSent rule. This rule is straightforward as it maps occurrences of the `signal_sent` event to occurrence of the `activated` event carrying the `name` of the signal (*i.e.,* the type of the signal occurrence carried by `signal_sent` event occurrences) as the value of their `id` parameter.

In total, the ECA rules of the subtyping relationship each required around 3 to 7 lines of Java code for the method bodies, as we were able to reuse the library for Esper-based subtyping relationships.

Fulfilling Req. 3. In this demonstration case, we defined ActivatableInterface as a common supertype of both the Arduino DSL and UML State Machines through subtyping relationships between ActivatableInterface and both ArduinoInterface and StateMachineInterface. In the process, we defined a library dedicated to Esper-based subtyping relationships, allowing us to keep the number of lines of Java code required to define each ECA rule of these relationship between 3 and 7 per method body.

Once defined, the subtyping relationships allow the previously defined reflective tools to be indiscriminately used with models conforming to either DSL, sending and receiving event occurrences from the ActivatableInterface interface in both cases. For example, the test runner can be used to check whether an Arduino model being a realization of a State Machine model behaves in the same expected way, by running the exact same test suite on both models, provided this test suite is designed with event occurrences from the ActivatableInterface interface. We were thus able to capitalize upon this and use tools with events from ActivatableInterface with models conforming to the Arduino DSL and with models conforming to UML State Machines, thereby fulfilling **Req. 3** for these two DSLs.

4.5.4 Summarized Results

In summary, the proposed metalanguage allowed us to define explicit behavioral interfaces for xDSLs, that specify how modelers and tools can soundly interact with running models.

Secondly, having such explicit behavioral interfaces, combined to the explicit API of the event manager allows the development of reflective tools reproducing essential features available in common executable modeling tools. This allows tools to be generic through reflection, as was demonstrated by using two such tools (a test runner and an event injection GUI) with two different xDSLs.

Finally, explicit behavioral interfaces allow subtyping relationships to be defined between them, which can in turn enable substitutability of xDSLs and thus genericity through abstraction. This then allows to define tools that are specific to a given behavioral interface but can in fact be used by any xDSL having this interface as a supertype. It also allows to substitute an xDSL by another one tailored for the task at hand (*e.g.,* analysis of state machines versus simulation plus code generation targeting Arduino platforms).

4.5.5 Threats to Validity

Internal Validity. As we are experienced in using the GEMOC Studio, we might have overlooked limitations to our approach that would make it hard to use for language engineers. Conducting a user study to assess the usefulness of our tools and the usability of our approach is an important direction of our future work.

External Validity. We verified that our approach yields its benefits for two xDSLs and two generic tools, which externally threatens the ability of our approach to be generalized to multiple DSLs and tools. However, the two selected xDSLs are relevant and representative of the languages supported by the approach as their abstract syntax is defined as a metamodel and their execution semantics is defined as a discrete-event operational semantics written in an object-oriented language and orchestrated by an execution engine. This indicates that the approach could be generalized to these languages, given their operational semantics provides the necessary granularity to enable the proper handling of events through calls to existing execution rules. In the opposite case, a refactoring of the operational semantics in accordance with the good practice of the separation of concerns is required.

Another threat to validity is that the DSLs used in the evaluation were implemented with interaction in mind, and thus presented the appropriate execution rules to correctly design their implementation relationships. This externally threatens the ability of our approach to be applied to any existing language without modifying it. However, the intent of our contribution is to provide a new way of designing DSLs and is thus geared towards the definition of new languages or the (possibly substantial) refactoring of existing ones, not towards opportunistic reuse of existing languages. As this is also an interesting potential application of the approach, we consider it as a future work direction.

4.5.6 Critical Discussion

While the approach works well for xDSLs whose conforming models are similar to state/transition systems, it presents some limitations when working with xDSLs that have time-related concepts, such as the Arduino DSL. More precisely, defining an ECA

rule similar to the `activate` rule for PushButton elements that allows for a customized duration between pressing and releasing a button (and more generally between two event occurrences) would require to be able to specify waiting times before specific event occurrences are sent by the event manager. Since the approach works at the language level, the issue is then to decide on a waiting time that will fit all conforming models, or to find a way to derive or define this waiting time on a model-by-model basis. Additionally, expressing time durations also requires a time unit, which could either be a generic unit (*e.g.,* execution steps) or a domain-specific one (*e.g.,* number of turns for a camshaft), specified at the language or model level, or even a real-time one such as seconds or milliseconds.

User-wise, the adoption of the approach by language engineers has an impact on how they work, and most notably on the way they design the execution semantics of their xDSL. Indeed, to enable the definition of events at any level of granularity, execution rules must be designed for a single task, and internal behavior needs to be clearly separated from potentially external behavior. This means that, when applying the approach on an existing DSL, some refactoring might be necessary to be able to define meaningful events. However, we believe that these requirements fit the good practice of the separation of concerns, advocating for methods to be dedicated to one precise task. Therefore, as long as language engineers implemented the execution semantics of their xDSL according to the separation of concern, little to no refactoring is necessary for adopting the approach.

Finally, the use of the Stop event discussed in Section 4.4.2.3 and the existence of a `run` event in both ArduinoInterface and StateMachineInterface indicate that a kind of "system" behavioral interface, dedicated to execution specific events (*e.g.,* starting and stopping the execution, pausing it, waiting) would be beneficial. This in turn hints at another purpose for behavioral interfaces, defined at the metalanguage level, which is worth investigating.

4.6 Summary

Interacting with running models is crucial for many tasks, ranging from automated testing, to communication between heterogeneous models, to manual interaction. Yet, providing interaction facilities for an xDSL is a tedious and error-prone task, which is hard to generalize due to the variety of shapes and forms xDSLs can take. To address this problem, we proposed an approach consisting in attributing *behavioral types* to xDSLs, under the form of explicit *behavioral interfaces* specifying the accepted and exposed events that can be used to interact with conforming models. In practice, language engineers do this by defining a reactive extension for their xDSL, which consists of the behavioral interfaces implemented, directly or transitively, by the xDSL, and of the corresponding *implementation* and *subtyping relationships*. This in turn enables the definition of generic tools for reactive DSLs, either through *reflection*, leveraging the explicit behavioral interface of DSLs, or through *abstraction*, leveraging the event abstraction hierarchies enabled by subtyping relationships to provide tools for families of DSLs. For the definition of this reactive extension, we provide a metalanguage to define behavioral interfaces. The

execution semantics of the defined interfaces is provided in part by the event manager, and in part by the implementation and subtyping relationships supplied by language engineers.

In the next chapter, we cover our contribution on offline behavioral analysis for xDSLs, which focuses on execution trace manipulation and analysis to evaluate, for example, the behavioral impacts of various environmental scenarios, or of structural changes to a model.

CHAPTER 5

Trace Comprehension Operators for Executable DSLs

Leveraging the work presented in the previous chapter, modelers can explore how the environment affect their reactive models through the behavioral types exposed by reactive DSLs. A prevalent mean of evaluating and analyzing this impact is the use of execution traces. For instance, modelers can execute their models under different scenarios sending varying sequences of event occurrences to the model, or execute different versions of their models with the same scenario. Then, analyzing and comparing execution traces allows modelers to verify that the actual behavior of their models matches the expected one. Yet, execution traces can quickly become overloaded with noisy or redundant data, hampering the comprehension of a particular behavior of the model.

To remedy this, we provide in this chapter an execution trace algebra allowing to filter out noisy and redundant data though the use of trace manipulation operators, thereby facilitating trace comprehension. We also provide two analysis operators for execution traces: a first operator to compare two traces and identify their differences, and a second one to extract the state graph corresponding to an execution trace.

The proposed operators offer a number of services in the envisioned ecosystem of interoperable tools. For instance, they can be used to steer online behavioral analysis by identifying key execution states such as bottlenecks. They can also be used to define test oracles by combining trace manipulation operators and either the trace comparison operator (with a reference trace) or the state graph operator (to compare the output graph with *e.g.,* a reference protocol). The operators can then help the modeler explore and understand potential test case failures.

The work presented in this chapter is the subject of a publication in the European Conference on Modeling Foundations and Applications [LBM+18].

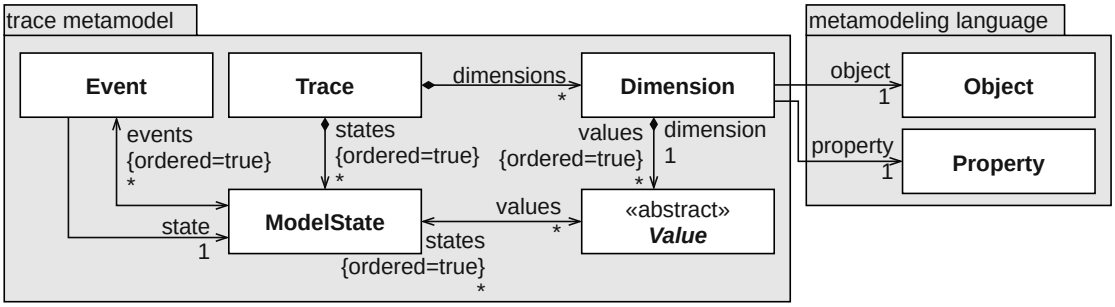


Figure 5.1: Generic trace metamodel.

5.1 Motivation

In this section, we first precisely scope the considered execution traces, and then introduce a State Machines DSL and an accompanying motivating example.

5.1.1 Considered Execution Traces

As traces are the main concept manipulated throughout this chapter, we define more precisely how they are structured in the form of a *generic trace metamodel* valid for any executable DSL. Figure 5.1 shows the proposed trace metamodel. The Trace metaclass, root of this metamodel, stores the sequence of model states as ModelState elements. Each ModelState element stores the events observed during the model state as Event elements. Each Event element points to the model state reached after the event occurrence. The Trace element also stores Dimension elements, each dimension containing the sequence of values taken by a dynamic slot. Thus, a Dimension element contains a sequence of Value elements, and points to an Object of the executed model and to a Property element of the metamodel it conforms to. Finally, each Value element points to the sequence of ModelState elements where this value was present, *i.e.,* the states during which the value of the dynamic slot referenced by its containing dimension remained the same. In accordance with the work on which this contribution builds [BMCB17, BCC+15], considering a trace as a set of dimensions is central to our approach, as it gives the possibility to efficiently manipulate the parts of a trace related to specific dynamic properties. We present examples of traces in the following subsection.

5.1.2 Motivating Example

The left part of Figure 5.2 shows the abstract syntax of an example of State Machines DSL, which is directly inspired from UML State Machines. A StateMachine contains at least one Region. A Region contains Vertex elements, which can be State elements or PseudoState elements. PseudoState elements are further refined into Initial, ExitPoint and

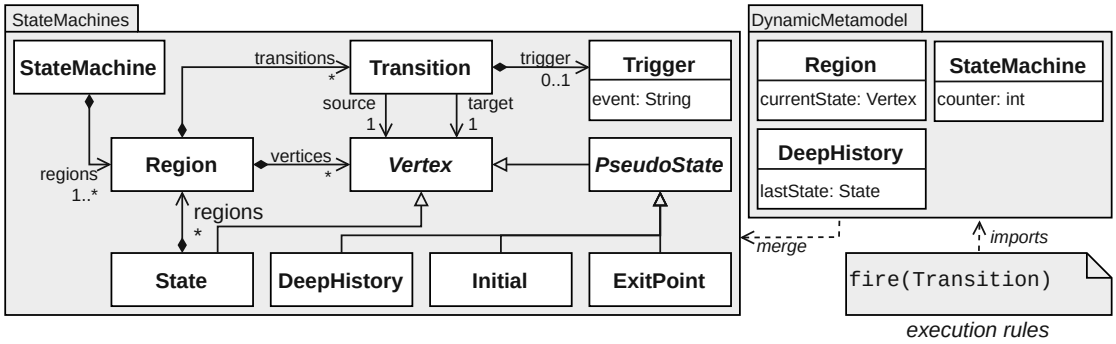
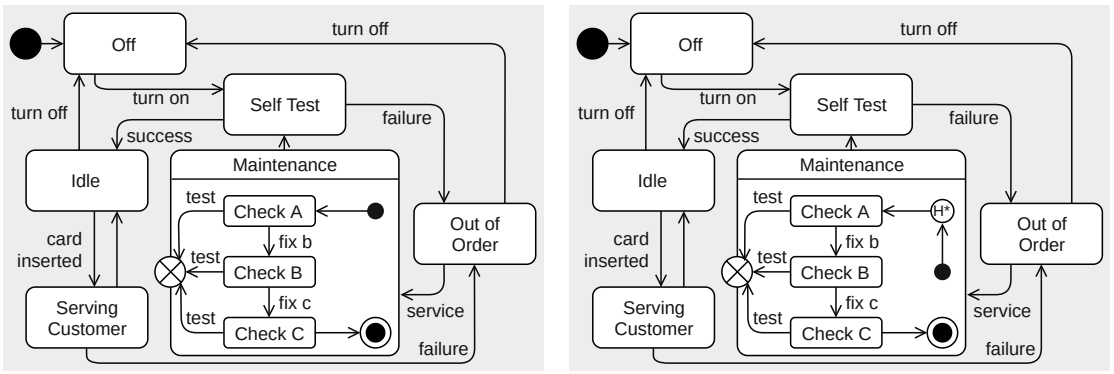


Figure 5.2: The State Machines executable DSL.



(a) Without history.

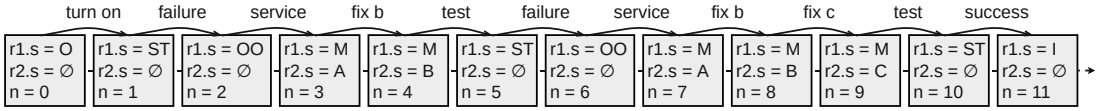
(b) With history.

Figure 5.3: Two ATM state machine variants.

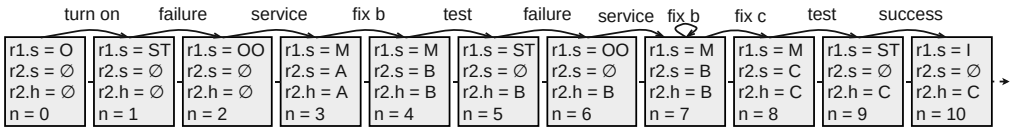
DeepHistory elements. Finally, a Region also contains Transition elements which point to a `source` and a `target` Vertex. Transition elements contain Trigger elements, which may each possess an `event`.

The upper right part of Figure 5.2 shows the metamodel of the operational semantics extending the abstract syntax with new dynamic properties: the `currentState` property in Region is used to track the current state of the Region during the execution, the `lastState` property in DeepHistory stores the last visited state in the owning Region element, and finally the `counter` counts the number of fired transitions during the execution. Finally, we consider that the only observable event of the DSL is the firing of Transition elements. Thus, the execution semantics only defines a `fire` event.

Figure 5.3 depicts two models conforming to the State Machines DSL shown in Figure 5.2. Both models represent the behavior of a *cash dispenser*, also called *ATM*, with states such as Idle or Serving Customer. Transitions represent how the ATM switches mostly between idling, maintenance and service states. The difference between the two models lies in the added deep history pseudostate in the region of the Maintenance state.



(a) Without history.



(b) With history.

Figure 5.4: Traces from the example models of Figure 5.3.

The semantics of the deep history pseudostate is that it stores the last visited state of its containing region and, when targeted by a transition, restores this state as the current state of the region. Adding such a pseudostate can thus affect greatly how the execution unfolds, and predicting the impact of such a change can be difficult.

Figure 5.4 shows two execution traces resulting from the execution of the models in Figure 5.3 with the following sequence of stimuli: *turn on*, *failure*, *service*, *fix b*, *test*, *failure*, *service*, *fix b*, *fix c*, *test*, *success*. Note that the second *fix b* event cannot be handled by model *b*, as model *b* is already in the CheckB state when the event is received: the event is thus discarded. In these traces, `r1` refers to the Region element owned by the state machine and `r2` to the Region element owned by the Maintenance state. The `s` property refers to the current state of a Region element, and the `h` refers to the last state of a DeepHistory element. Finally, the `n` property refers to the counter of fired transitions of the state machine. The name of the states are abbreviated for space reasons.

By looking at the execution traces shown in Figure 5.4, we can glimpse that if we were ignoring the dimensions `counter` and `lastState`, then many similarities between the two traces could be found. For instance, the states ⟨Maintenance, CheckC, 9⟩ and ⟨Maintenance, CheckC, CheckC, 8⟩ would then be equivalent. Even in a single trace, the value of the `counter` is different in each model state, which make it hard to identify possible cycles encountered in the states of the State Machine. For instance, the trace shown in Figure 5.4b features a cycle that can only be detected if the `counter` dimension is ignored: ⟨Maintenance, CheckB, 4⟩ and ⟨Maintenance, CheckB, 8⟩ are two states part of this cycle.

In summary, even with small models with little dynamic information, and with only small changes between model variants, it is already challenging to understand and to compare the resulting execution traces. A similar observation could be made if small changes were made to the semantics of the considered DSL, or to the stimuli of the considered execution scenario. Scaling up to complex models with more dynamic information makes it even harder to extract meaningful information from execution traces. In this context, to ease the task of understanding execution traces, our contribution is a set of four *trace comprehension operators*. We give a brief presentation of these operators in Section 5.2, before defining them formally in Section 5.3.

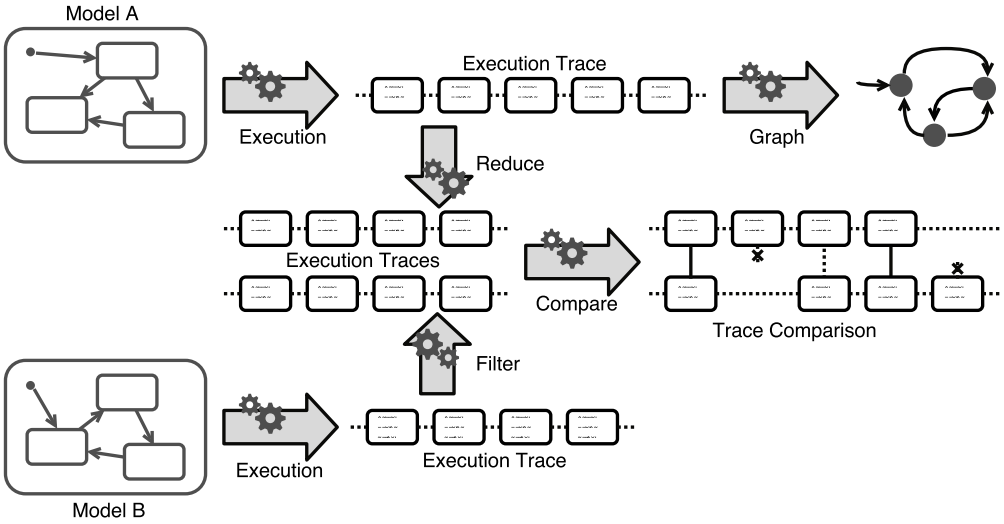


Figure 5.5: Overview of a possible workflow using all four proposed operators.

5.2 Approach Overview

In this section, we present an overview of the contribution of this chapter, *i.e.,* four different and complementary trace comprehension operators. Figure 5.5 summarizes the application context including the inputs and outputs of the different operators. On the left, two behavioral models named A and B are shown, and a trace is obtained for each of their executions. Then, four different operators can be used to manipulate the obtained traces:

- The *Filter* operator takes an execution trace as input and produces a refined version of the input execution trace as output. It removes a selected set of dimensions (see Section 5.1.1 for the definition of dimension) from the input trace, which results in a simplified trace that only reflects the evolution of a subset of the model state. Note that *Filter* does not change the amount of model states in the trace, and only changes the content of each model state.
- The *Reduce* operator also takes an execution trace as input and produces a refined version of the input execution trace as output, where each subsequence of successive identical model states is merged into a single model state. *Reduce* is particularly useful when applied after the *Filter* operator when the only differences between the states of a sequence of successive states were found in the dimensions that were filtered out.
- The *Compare* operator takes two execution traces as input and produces a trace difference model as output. This difference model highlights all the changes that occurred between the first trace and the second one: which states were added,

removed, or substituted by other states. Such comparison can be used to better understand the impact of a design change on the trace resulting from the execution.

– The *Graph* operator takes an execution trace as input and produces a state graph as output. This state graph is a representation of all different model states reached during the execution. Among other benefits, such higher-level view provides a better global understanding of the execution, and can highlight cycles and bottleneck states.

The middle and right part of Figure 5.5 show a typical workflow where the traces obtained from the models are simplified through the use of the *Filter* and *Reduce* operators, before being used as input for the *Graph* and *Compare* operators. In the following section, we provide a formal specification of these four operators.

5.3 Operators for Execution Trace Comprehension

In this section we present our contribution, *i.e.,* a set of four trace comprehension operators. Figure 5.6 summarizes graphically all proposed operators using abstract examples, and will be used throughout the section to illustrate the operators. In what follows, we first formally define what is a execution trace, then we provide a formal definition of each trace comprehension operator.

5.3.1 Execution Trace Formalization

In order to give a formal definition of operators that manipulate execution traces, we must first formally define the concept of trace. In the remainder of this chapter, we denote T the set of all execution traces, and V the set of all observable values during an execution.

Definition 8 (Trace)**.** A trace is a tuple $\langle S, D, E_<, val, step \rangle$ where:

– S is the set of model states of the execution trace.

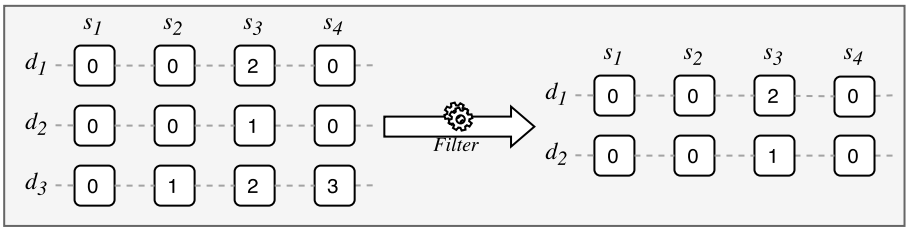
– D is the set of dimensions of the execution trace.

– $E_< = (E, <_E)$ is the totally ordered set of events that occurred during the execution where, $\forall e_1, e_2 \in E, e_1 <_E e_2$ if e_1 happens before e_2.

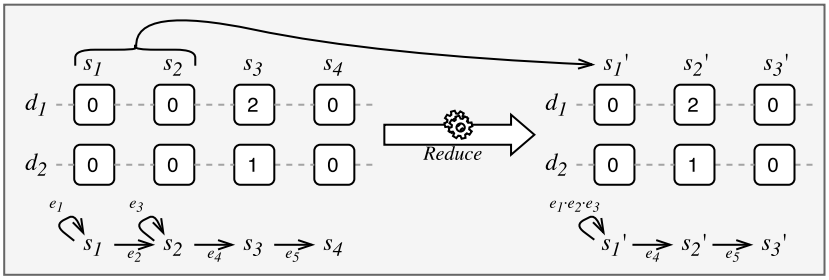
– $val : (S \times D) \rightarrow V$ is the function mapping a model state and a dimension to a value. Using val, we define a state equivalence relation $Eq \subseteq S \times S$ as $(a, b) \in Eq \Leftrightarrow \forall d \in D, val(a, d) = val(b, d)$, denoted $a \equiv b$.

– $step : E_< \rightarrow (S \times S)$ is the function mapping an event to a starting an ending state. Note that an event *can* have the same starting and ending state, which means that the model state did not change due to the event occurrence. We denote:

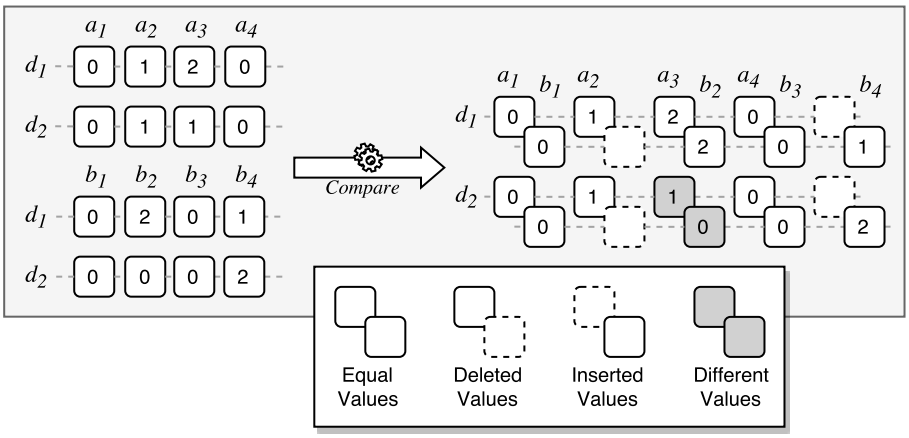
  – $a \xrightarrow{e} b$ the fact that $step(e) = (a, b)$,



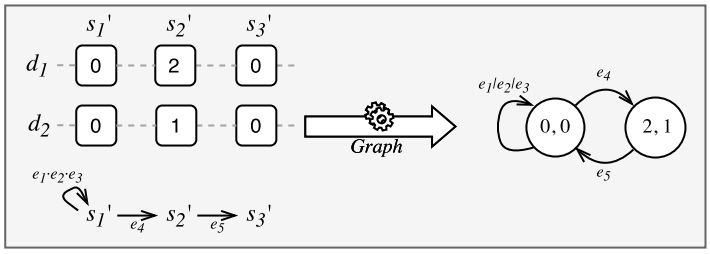
(a) *Filter* operator, which removes dimensions from a trace.



(b) *Reduct* operator, which factorizes redundant states.



(c) *Compare* operator, which produces a trace difference.



(d) *Graph* operator, which extracts a state graph from a trace.

Figure 5.6: Graphical summary of all four trace comprehension operators.

– $a \xrightarrow{*} b$ the fact that *step* can lead from a to b with a sequence of events, *i.e.*:

$$\exists e \in E_<, a \xrightarrow{e} b \vee \exists n \in \mathbb{N}, \exists e_1, ..., e_n \in E_<, \exists s_1, ..., s_{n-1} \in S,$$
$$a \xrightarrow{e_1} s_1 \xrightarrow{e_2} ... \xrightarrow{e_{n-1}} s_{n-1} \xrightarrow{e_n} b \wedge \forall i \in]1; n], e_{i-1} <_E e_i,$$

– $a \rightarrow b$ the fact that $a \neq b \wedge \exists e \in E_<, a \xrightarrow{e} b$, *i.e.*, a directly precedes b,

– the total order on events $<_E$ and the *step* function are combined into a total order $<_S$ over the states, which is defined as: $\forall a, b \in S, a <_S b \Leftrightarrow a \xrightarrow{*} b$. We denote $s = S_i$ the fact that $|\{s' \in S : s' <_S s\}| = i$, *i.e.*, the fact that s is the i-th state of the trace.

Example 1. Using only natural integer values (*i.e.*, $V = \mathbb{N}$), and events e_i ordered by their index i, let t_{ex} be an execution trace conforming to Definition 8:

$$\begin{aligned} t_{ex} \; &= \langle \{s_1, s_2, s_3, s_4\}, \{d_1, d_2, d_3\}, \{e_1, e_2, e_3, e_4, e_5\}, val, step \rangle \\ \text{where } & val(s_1, d_1) = 0 \quad val(s_2, d_1) = 0 \quad val(s_3, d_1) = 2 \quad val(s_4, d_1) = 0 \\ & val(s_1, d_2) = 0 \quad val(s_2, d_2) = 0 \quad val(s_3, d_2) = 1 \quad val(s_4, d_2) = 0 \\ & val(s_1, d_3) = 0 \quad val(s_2, d_3) = 1 \quad val(s_3, d_3) = 2 \quad val(s_4, d_3) = 3 \\ \text{and } & s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4 \end{aligned}$$

5.3.2 Dimension Filtering

When an operational semantics introduces a large amount of dynamic properties, or when the executed model is very large, an execution trace may contain a large amount of dimensions to grasp. Yet, understanding specific aspects of the behavior might only require looking of a specific subset of dimensions of interest. For this purpose, our first operator is called *Filter* (see Figure 5.6a), and aims at removing dimensions out of a trace in order to simplify it. This operator is in fact an abstraction operator on the model states contained in the trace.

Definition 9 (*Filter*)**.** Given an input trace $\langle S, D, E_<, val, step \rangle$ and an input set of dimensions I, the *Filter* operator is defined as:

$$\begin{array}{rccc} Filter: & (T \times \mathcal{P}(D)) & \rightarrow & T \\ & (\langle S, D, E_<, val, step \rangle, I) & \mapsto & \langle S, D', E_<, val', step \rangle \end{array}$$

where $D' = D \setminus I$ and $val' : S \times D' \rightarrow V$ is defined as $val'(s, d') = val(s, d')$.

Example 2. We apply *Filter* to the trace t_{Ex} and dimension d_3 from Example 1:

$$Filter(t_{Ex}, \{d_3\}) = \langle \{s_1, s_2, s_3, s_4\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step \rangle$$

$$\begin{aligned}\text{where } & val'(s_1, d_1) = 0 \quad val'(s_2, d_1) = 0 \quad val'(s_3, d_1) = 2 \quad val'(s_4, d_1) = 0 \\ & val'(s_1, d_2) = 0 \quad val'(s_2, d_2) = 0 \quad val'(s_3, d_2) = 1 \quad val'(s_4, d_2) = 0 \\ \text{and } & s_1 \xrightarrow{e_1} s_1 \quad s_1 \xrightarrow{e_2} s_2 \quad s_2 \xrightarrow{e_3} s_2 \quad s_2 \xrightarrow{e_4} s_3 \quad s_3 \xrightarrow{e_5} s_4\end{aligned}$$

Note that $s_1 \equiv s_2$ and $s_1 \rightarrow s_2$, *i.e.,* s_1 and s_2 are two identical successive model states. The next operator will enable the merging of these states to obtain a more compact trace, *i.e.,* where a state is always different from its preceding state.

5.3.3 Trace Reduction

When using a trace recorder that always records the model state at each occurring observable event without checking if the state has changed, or when using the *Filter* operator introduced above, a trace may contain successive equivalent states which are redundant and can be considered as superfluous data. This phenomenon is also known as stuttering [GV90]. To simplify such traces, we propose an operator *Reduce* (see Figure 5.6b) which merges such successive equivalent states while preserving the behavior depicted by the trace.

Definition 10 (*Reduce*)**.** The *Reduce* operator is defined as:

$$\begin{array}{rccc} Reduce: & T & \rightarrow & T \\ & \langle S, D, E_<, val, step\rangle & \mapsto & \langle S', D, E_<, val', step'\rangle \end{array}$$

where:

- S' is the set of sets of successive equivalent states of S, *i.e*:

$$S' = \{s \in \mathcal{P}(S) : \forall a \in s, \forall b \in S, a \equiv b \wedge a \rightarrow b \Rightarrow b \in s\}$$

- $step' : E_< \rightarrow (S' \times S')$ is defined as: $step'(e) = \langle A, B\rangle \Leftrightarrow step(e) \in (A \times B)$
- $val' : (S' \times D) \rightarrow V$ is defined as $val'(B, d) = val(a, d)$ for any $a \in B$

Hence, each output state of S' is composed of (and thus replaces) a set of equivalent successive states of S, and both $step'$ and val' are adjusted accordingly.

Example 3. Resulting trace from $Reduce(Filter(T_{Ex}, \{d_3\}))$.

$$\begin{aligned} & Reduce(Filter(T_{Ex})) = \\ & \quad \langle \{s'_1 = \{s_1, s_2\}, s'_2 = \{s_3\}, s'_3 = \{s_4\}\}, \{d_1, d_2\}, \{e_1, e_2, e_3, e_4, e_5\}, val', step'\rangle \\ & \qquad \text{where } val'(s'_1, d_1) = 0 \quad val'(s'_2, d_1) = 2 \quad val'(s'_3, d_1) = 0 \\ & \qquad\qquad val'(s'_1, d_2) = 0 \quad val'(s'_2, d_2) = 1 \quad val'(s'_3, d_2) = 0 \\ & \qquad\quad \text{and } s'_1 \xrightarrow{e_1} s'_1 \quad s'_1 \xrightarrow{e_2} s'_1 \quad s'_1 \xrightarrow{e_3} s'_1 \quad s'_1 \xrightarrow{e_4} s'_2 \quad s'_2 \xrightarrow{e_5} s'_3 \end{aligned}$$

The soundness of *Reduce* can easily be proven, *i.e.,* the fact that two successive states of S' cannot be equivalent, and that one state of S is only mapped to a single state of S'. These properties can be rephrased as two theorems:

Theorem 1. $Reduce(T) = \langle S', \_, \_, \_, \_ \rangle \Rightarrow \forall s_1, s_2 \in S', s_1 \rightarrow s_2 \Rightarrow s_1 \not\equiv s_2$

Theorem 2. $Reduce(T) = \langle S', \_, \_, \_, \_ \rangle \Rightarrow \bigcap_{s \in S'} s = \emptyset$

5.3.4 Trace Comparison

As understanding a single execution trace is already a difficult task, grasping the differences between two execution traces is even more challenging and error-prone. To address this problem, we propose a *Compare* operator that produces a *trace difference* showing the similarities and dissimilarities between two traces. Note that since traces may come from different models (*e.g.,* an original and a revised one), each trace may possess its own set of dimensions, hence *Compare* requires an explicit mapping between the dimensions of the first and second traces.

Our comparison procedure relies on the notorious *Levenshtein distance* [Lev66], which is an operator counting the minimal number of insertion, deletion or substitution operations required to transform one string into another. For instance, the Levenshtein distance between "*S*TR**ING**" and "TR**A**C**E**" is four, which is computed by summing the number of insertions in *italics* and of substitutions in **bold**. While the output of the *Levenshtein distance* is an integer, computing this distance requires computing all the distances between all the possible prefixes of the input strings (*i.e.,* substrings starting with the first character). It is then possible to infer from all these distances the exact set of insertions, deletions or substitutions required to transform the first string into the second string, which is the kind of information we require to construct a trace difference. For our work, we adapted the Levenstein distance to compare traces instead of strings, where model states play the role of characters, which can be compared using the equivalence relation.

Definition 11 (Levenshtein distance on traces)**.** The Levenshtein distance between two traces $T_1 = \langle A, \_, \_, \_, \_ \rangle$ and $T_2 = \langle B, \_, \_, \_, \_ \rangle$ is given by $lev_{T_1,T_2}(|A|, |B|)$ where:

$$lev_{T_1,T_2}(i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0, \\ min \begin{cases} lev_{T_1,T_2}(i-1,j) + 1 \\ lev_{T_1,T_2}(i,j-1) + 1 \\ lev_{T_1,T_2}(i-1,j-1) + 1_{A_i \not\equiv B_j} \end{cases} & \text{otherwise} \end{cases}$$

Where $1_{A_i \not\equiv B_j}$ equals 0 when $A_i \equiv B_j$, and equals 1 otherwise.

As we can see, to obtain the Levenstein distance $lev_{T_1,T_2}(|A|, |B|)$, we rely on a recursive operator $lev_{T_1,T_2}(i,j)$ which computes the distance between the subsequence of states $[0,i]$ of T_1 and the subsequence of states $[0,j]$ of T_2. These distances can be used to infer the insertions, deletions and substitutions required to go from the first trace to the second. In that goal, we define the following notations on top of *lev*:

– $in_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i,j-1) + 1$,

- $del_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j)+1$,
- $subst_{T_1,T_2}(i,j)$ denotes $lev_{T_1,T_2}(i,j) = lev_{T_1,T_2}(i-1,j-1)+1$.

Using this $lev_{T_1,T_2}(i,j)$ through these notations, we can define the *Diff* operator which produces a unique set containing states of T_1 that were deleted, states of T_2 that were inserted, and pairs of states from T_1 and T_2 that were substituted.

Definition 12 (*Diff*)**.** We define the union set of inserted, deleted and pairs of substituted states identified as part of a Levenshtein distance computation as $Diff_{T_1,T_2} = DiffRec_{T_1,T_2}(|A|,|B|)$, where:

$$DiffRec_{T_1,T_2}(i,j) = \begin{cases} DiffRec_{T_1,T_2}(0,0) = \emptyset & \\ DiffRec_{T_1,T_2}(i,j-1) \cup B_j & \text{if } in_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j) \cup A_i & \text{if } del_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) \cup \{A_i, B_j\} & \text{if } subst_{T_1,T_2}(i,j) \\ DiffRec_{T_1,T_2}(i-1,j-1) & \text{otherwise} \end{cases}$$

Finally, we define *Compare* as a trivial projection of the output of the *Diff* operator into a tuple that separates insertions, deletions and substitutions in three different sets.

Definition 13 (*Compare*)**.** Given two traces $T_1 = \langle A, \_, D_1, \_, \_\rangle$ and $T_2 = \langle B, \_, D_2, \_, \_\rangle$ and a mapping $M \subseteq \mathcal{P}(D_1 \times D_2)$, the *Compare* operator is defined as:

$$\begin{array}{rccc} Compare: & T \times T \times (D_1 \times D_2) & \rightarrow & \mathcal{P}(B) \times \mathcal{P}(A) \times \mathcal{P}(A \times B) \\ & (T_1, T_2, M) & \mapsto & \langle In, Del, Subst\rangle \end{array}$$

where $In = Diff_{T_1,T_2} \cap B$, $Del = Diff_{T_1,T_2} \cap A$ and $Subst = Diff_{T_1,T_2} \cap (A \times B)$.

Note that while the comparison results are unordered, this does not prevent the presentation of the comparison result in a human-readable way. This can be done by iterating over the states of both traces in parallel, and looking for them in the trace difference. For instance, Figure 5.6c was obtained from the trace difference obtained with *Compare* containing $\langle\{b_4\},\{a_2\},\{\langle a_3,b_2\rangle\}\rangle$ using the following reasoning:

- a_1 and b_1 are absent from the result: hence all their values are equal (first column).
- a_2 is not contained in a pair: hence it has been deleted from t_1 (second column).
- a_3 and b_2 are contained in a pair: hence some values are different from a_3 to b_2. These values can be identified by iterating over the dimension pairs that are part of the provided matching (third column).
- a_4 and b_3 are absent from the result: hence all their values are equal (fourth column).
- b_4 is not contained in a pair: hence it has been inserted in t_2 (fifth column).

5.3.5 State Graph Extraction

For each model state in an execution trace, there may be other *equivalent* model states scattered over the trace, which means that the execution is going back to this state several times during the execution. However, the sequential nature of a trace makes it difficult to grasp such information and understand the possible *cycles* in the execution trace. To provide a better understanding of the encountered model states, we propose the last operator called *Graph* (see Figure 5.6d). This operator creates a directed graph from a trace. Each vertex in the graph is mapped to a set of equivalent states of the trace, and each event adds an edge between the vertexes containing its source and target states, if such an edge does not already exists. These edges also carry the set of events that caused their existence.

Definition 14 (*Graph*)**.** Let G be the set of all directed graphs. The *Graph* operator is defined as:

$$\begin{array}{rccc} Graph: & T & \rightarrow & G \\ & \langle S, D, E_<, val, step\rangle & \mapsto & \langle V, A\rangle \end{array}$$

where:

- V is the set of vertices, with $V = \{s \in \mathcal{P}(S) : \forall a, b \in s, a \equiv b\}$
- A is the set of directed edges, with $A = \{\langle v_1, Events, v_2\rangle \in V \times \mathcal{P}(E) \times V : \forall e \in Events, \exists a \in v_1, \exists b \in v_2, a \xrightarrow{e} b\}$

Example 4. Resulting graph from $Graph(Filter(T_{Ex}, \{d_3\}))$

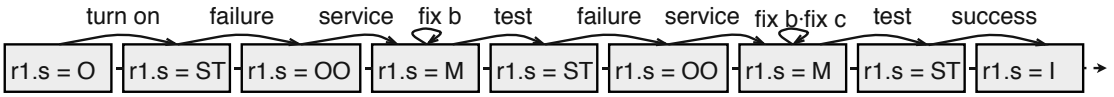
$$Graph(Filter(T_{Ex})) = \langle \{v_1 = \{s_1, s_2, s_4\}, v_2 = \{s_3\}\}, \{(v_1, \{e_1, e_2, e_3\}, v_1), (v_1, \{e_4\}, v_2), (v_2, \{e_5\}, v_1)\}\rangle$$

5.4 Evaluation

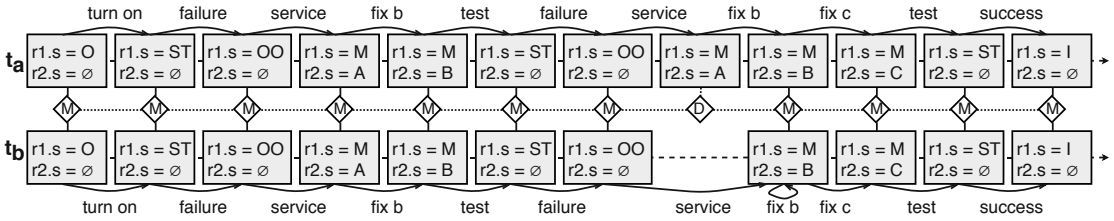
In this section, we present how we validate the approach using the motivating example conforming to the State Machine DSL from Section 5.1.2.

To evaluate the contribution of this chapter, we demonstrate the usefulness of the proposed operators to understand the execution traces of the State Machine models previously shown as a motivation in Section 5.1.2. We recall that the models were depicted in Figure 5.3, and the considered execution traces in Figure 5.4. Figure 5.7 shows different applications of the four operators to the two execution traces, most of them by combining the use of multiple operators. We explain below how the results help better understand the traces.

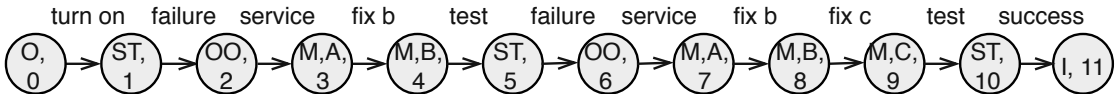
***Filter* and *Reduce*.** To obtain the trace shown in Figure 5.7a from t_a, we first apply *Filter* on the `counter` and Maintenance.`currentState` dimensions, then we apply *Reduce*. We choose to filter the `counter` dimension because it changes at each state and thus hampers further cycle analysis or trace comparison, and the Maintenance.`currentState`



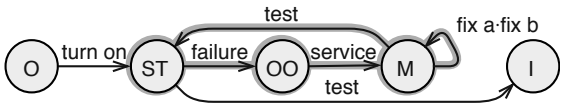
(a) *Reduce*(*Filter*(t_a, {`counter`, Maintenance.`currentState`}))



(b) *Compare*(*Filter*(t_a, {`counter`}), *Filter*(t_b, {`counter`, Maintenance.`lastState`}}))



(c) *Graph*(t_a)



(d) *Graph*(*Reduce*(*Filter*(t_a, {`counter`, Maintenance.`currentState`})))

Figure 5.7: Various applications of the operators on the traces from in Figure 5.4.

in order to hide the internal working of the Maintenance hierarchical state. The result is a more high-level trace which only focuses on the information of interest, *i.e.,* which states of the main state machine were visited. This demonstrates that the *Filter* and *Reduce* can be used both to get rid of noisy data (*e.g.,* the `counter`), and to modulate the level of detail featured in a trace my removing undesired dimensions (*e.g.,* Maintenance.`currentState`).

Compare. To obtain the trace difference shown in Figure 5.7b, we first apply *Filter* on the `counter` dimension on t_a and t_b and on the Maintenance.`lastState` dimension on t_b, then we apply *Compare* on the resulting traces. The mapping of dimensions provided to *Compare* is not shown, as it is trivial except for the deep history dimension which has no match. Figure 5.7b shows us that both traces align almost perfectly—except for a deleted state from one trace to the other—which was difficult to notice simply by looking at the original traces from Figure 5.4. Note that this result is only possible because the traces were filtered before the comparison, since comparing unfiltered traces would not find much similarities because of the `counter` property. This demonstrates that the *Compare* operator, especially when combined with *Filter* and *Reduce*, can effectively help to understand subtle behavioral differences induced by design choices.

Graph. To obtain the graph shown in Figure 5.7c, we directly applied *Graph* on the original t_a trace. We can observe that the resulting graph is of little interest as it takes the form of a sequence identical to t_a, which is mostly due to the incremented `counter`. However, in Figure 5.7d, we first applied the *Filter* operator on t_a to filter out the `counter` and Maintenance.`currentState` dimensions, followed by the *Reduce* operator. Applying *Graph* on the resulting trace shows us a better overview of the states visited during the execution. In particular, we can observe a *cycle* in the visited states, highlighted in gray. This demonstrates that the *Graph* operator, especially when combined with *Filter* and *Reduce*, can effectively help understanding which model states were visited in the execution, and which cycles can be observed between model states.

Additional material. Our companion web page[1] extends this evaluation with more complex models conforming to a real world DSL called ThingML. These chained operator applications illustrate both the complementarity of the operators and their usefulness to understand and analyze execution traces.

5.5 Summary

Execution traces obtained from the execution of behavioral models are essential sources of feedback, for instance to perform trade-off analyses or to evaluate how models react to their environment in different scenarios. Yet, due to the presence of noisy data in execution traces, it is often difficult for a modeler to understand how design changes and inputs from the environment impact the obtained execution traces and thus the behavior of the model. To address this problem, we proposed a set of formally defined *trace comprehension operators* comprising trace *manipulation* and trace *analysis* operators. Using trace manipulation operators, modelers can filter out the dimensions tracing noisy data from their execution traces, and reduce them by merging the subsequent equivalent states, thereby providing a trimmed-down view that facilitates comprehension. Then, through trace analysis operators, modelers can compare execution traces and compute an alignment model to perform trade-off analysis or compare different execution scenarios. Modelers can also extract a state graph from an execution trace, providing a data structure and visualization better suited for cycle and bottleneck state detection. An appropriate combined use of the trace manipulation and analysis operators also allows to compare the behavior of two similar but different models by filtering out any dimension that is not shared by the models.

In the next chapter, we cover our contribution to the online behavioral analysis of executable models, focusing more specifically on the runtime monitoring of temporal properties expressed at the domain level.

[1] `http://gemoc.org/ecmfa18`

CHAPTER 6

Runtime Monitoring for Executable DSLs

While the trace comprehension operators presented in the previous chapter propose offline behavioral analysis facilities for executable models, similar facilities for online behavioral analysis are also often required, in particular in the case of reactive models. For example, when debugging a reactive model, modelers need to be able to easily suspend the execution on specific circumstances, *e.g.,* to send an event occurrence to the running model. The same is true when defining test cases for reactive models: particular event occurrences must be sent at specific times during the execution. Detecting such circumstances can be done through the use of temporal properties evaluated during the execution. However, such properties are usually defined either in low-level languages such as LTL, thus requiring an expertise that cannot be asked of modelers, or with property languages dedicated to specific domains, that cannot be reused easily for other domains.

To circumvent this, we propose in this chapter an accessible yet expressive temporal property language allowing to define temporal properties at the domain-level. The properties defined with this language are then translated to runtime monitors, and deployed to a backend where they can be evaluated during the execution of a model. Other tools can register as observers of these runtime monitors to receive a notification when a verdict is reached on their corresponding temporal property.

The combination of the temporal property language and runtime monitoring backend fulfills important services in the ecosystem of interoperable tools. For instance, it allows modelers to formulate breakpoint conditions using domain concepts, which in turn spares them a step-by-step execution when debugging, while still providing them with the necessary control over the execution to reach a precise execution state. The temporal property language also enables the definition of advanced test oracles, the validation of specified temporal properties acting as checkpoints along the execution of a test case.

The work presented in this chapter is the subject of a publication in the European Conference on Modeling Foundations and Applications [LJB+20].

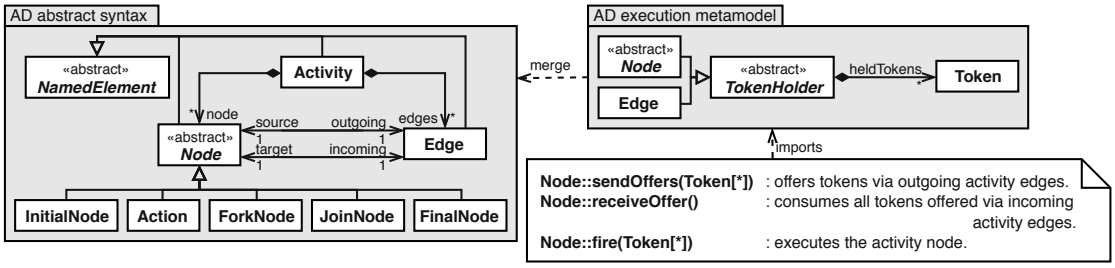


Figure 6.1: *Activity Diagram* DSL with an operational semantics.

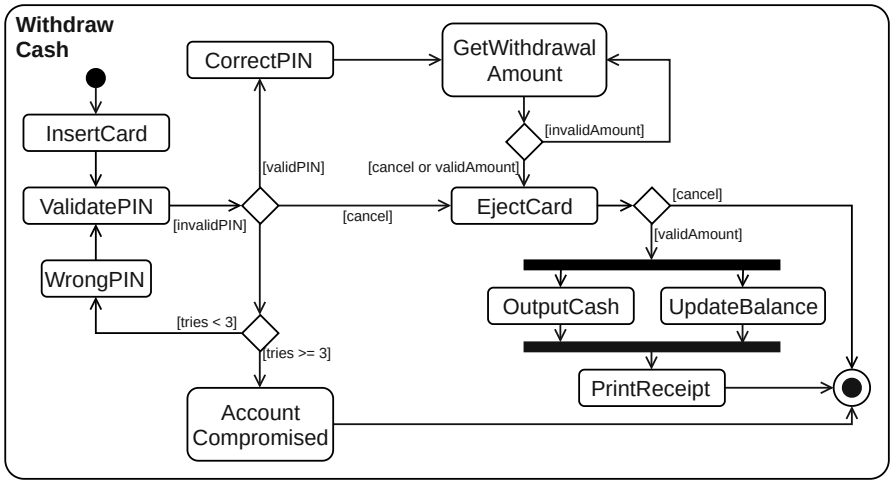


Figure 6.2: The *Withdraw Cash* Activity of an ATM.

6.1 Motivation

We first introduce the *Activity Diagram* DSL that we will use as a running example throughout this chapter. Figure 6.1 shows the definition of this DSL. This example is a simplified version of the part of fUML [Obj13b] related to the control flow of activities. At the top left, the abstract syntax defines an Activity as a set of inter-connected Node and Edge objects, with several types of nodes. InitialNode and FinalNode mark the beginning and the end of the Activity. A ForkNode starts concurrent execution branches, which can be joined back in a JoinNode. An Action represents an opaque action realized in the process. At the top right, the execution metamodel defines what is the model state of an activity by introducing a new metaclass called TokenHolder. This metaclass enables Node and Edge elements to contain Token elements. Lastly, the operational semantics of the DSL is defined by a set of execution rules, and is broadly based on a token flow starting in the initial node and ending in the final node. Three execution rules for the Node metaclass are shown at the bottom: *receiveOffers* and *sendOffers* that respectively take and put tokens from edges, and *fire* that (1) triggers *receiveOffers*, (2) performs the opaque action of the node, (3) triggers *sendOffers*.

Figure 6.2 shows an example Activity conforming to the *Activity Diagram* DSL shown in Figure 6.1. The shown activity is a simplified process of withdrawing cash at an ATM. First, the card is inserted into the ATM. Then, the PIN entered by the user is checked. If invalid, a new PIN is requested and checked, until the correct PIN is entered, the number of failed attempts reaches 3, or the user cancels the process. If no successful attempt is made in 3 tries, the card is swallowed and the account flagged as compromised. On a successful attempt, the user is asked for a withdrawal amount until she or he enters a valid amount (with regard to her or his account balance, the amount being a multiple of 10, etc.). Alternatively, the user can choose to cancel her or his withdrawal. Should the user choose to cancel the process at the PIN validation stage or at the withdrawing stage, the card is ejected and the activity ends. However, if a valid amount has been entered, the ATM first ejects the card, then concurrently outputs the correct amount of cash and updates the account balance of the user. Finally, the ATM prints a receipt indicating the amount withdrawn and the new account balance.

There are several temporal properties one might want to monitor on such an activity. For instance, it would be interesting to monitor whether the card always ends up being ejected once the correct PIN has been entered (**`P1`**). Another interesting property that can be monitored is that the card is always ejected before cash is distributed, to prevent people from forgetting their card in the ATM (**`P2`**). A last property one can monitor is that an invalid PIN is not entered more than two times between the moment a card is inserted and a correct PIN is entered (**`P3`**).

One possibility for monitoring these properties is to instrument the execution semantics of the *Activity Diagram* DSL. However, this requires to design an instrumentation technique that is tightly coupled to the *Activity Diagram* DSL, and to the metalanguage used to define its execution semantics. This coupling would hamper the potential reuse of the resulting instrumentation technique, requiring an adaptation for each new xDSL and/or metalanguage to support. In this chapter, we propose an alternate solution that can be tailored to the metalanguages used to define the abstract syntax and the execution semantics of xDSLs.

In Section 6.2 we provide an overview of the proposed approach, before detailing its inner workings in Sections 6.3 and 6.4.

6.2 Approach Overview

In this section, we provide an overview of the approach from the modeler's point of view, both at design time and at runtime. This overview is illustrated by Figure 6.3.

6.2.1 Design Time

At design time, modelers define temporal properties using the *temporal property language* proposed as part of the approach, as shown at the top of Figure 6.3. Modelers first specify what are the states of interest to detect in the execution, and then define the temporal pattern that must be observed with these states during the execution for the property to

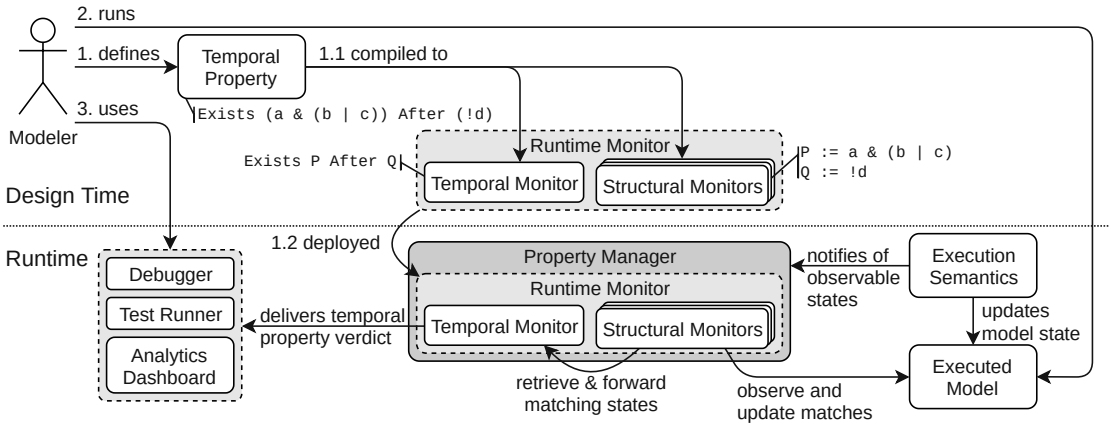


Figure 6.3: Overview of the approach.

be satisfied. For the definition of the states of interest, the approach relies on structural patterns written in Viatra Query Language (VQL) [BURV11]. VQL is a declarative graph query language which allows to directly express queries using the domain-specific concepts expressed in metamodels. In [BSVV18, BSVV19], Búr *et al.* successfully used VQL for the runtime monitoring of distributed safety properties over models@runtime. While our proposed approach includes the runtime monitoring of non-safety property, although not in a distributed way, the successful use of VQL emphasizes its relevance and motivates our use of this technology to realize our approach. For the definition of the temporal pattern, the approach relies on the PSPs, which makes it easy to express the most-used kinds of temporal properties [BGPS12]. The resulting abstract syntax of the proposed property language is presented in Section 6.3.1.

The compiler will then process a temporal property by separating its structural concern of from its temporal concern, and respectively produce *structural monitors* that are each able to detect when a specific structural pattern is encountered, and a Complex Event Processing (CEP)-based *temporal monitor* able to reason over time on states detected by structural monitors. An overview of the execution semantics of the proposed property language is presented in Section 6.3.2, and a focus on the translation scheme used to derive runtime monitors from temporal patterns is given in Section 6.4.

6.2.2 Runtime

At runtime, the runtime monitors derived from temporal properties are deployed into a property manager, as illustrated at the bottom of Figure 6.3. This property manager handles the connection between the structural and temporal monitors, the execution semantics, the running model and the various property listeners (*e.g.,* debugger, test runner, analytics dashboard, etc.).

At the start of the execution, the structural monitors are configured to update their matches whenever changes occur in the state of the running model, as shown on the lower left part of Figure 6.3. These matches are however only retrieved when an observable

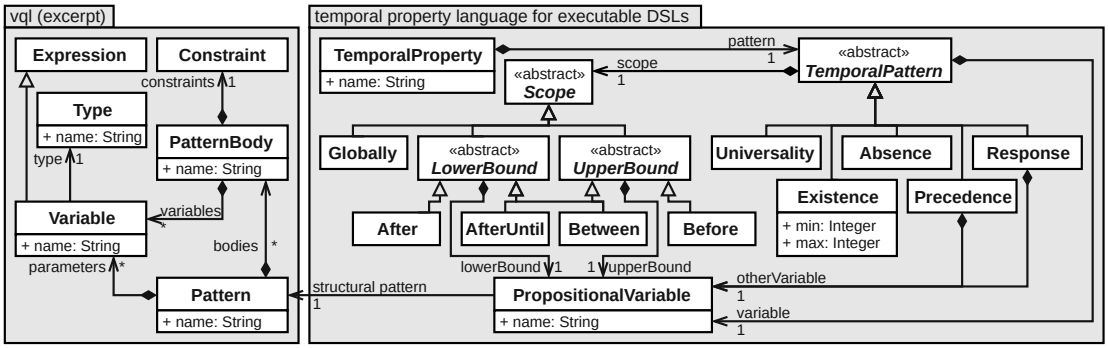


Figure 6.4: Abstract syntax of the proposed temporal property language.

model state is reached, of which the property manager is notified by the execution semantics of the DSL, as earlier explained in Section 2.4. This is to avoid evaluating the property on inconsistent model states.

Pattern matches are then sent to the temporal monitor, as shown on the lower middle part of Figure 6.3. The temporal monitors take these new matches (or absence thereof) into consideration to evaluate whether the property is satisfied or violated, or if no verdict can be rendered yet. Then, in case a final verdict is rendered, it is delivered to the property manager, which relays it by notifying the potential property listeners, as shown on the right lower part of Figure 6.3.

How structural and temporal monitors work together with the property manager is described in Section 6.3.2.

6.3 Temporal Property Language for xDSLs

In this section, we present our temporal property language for xDSLs, and how this language can be used for the runtime monitoring of executable models. The proposed language is based on VQL for expressing structural patterns, and on the PSPs for expressing temporal patterns. We first define the abstract syntax of the language, and then give an overview of the translation scheme used to derive runtime monitor from properties defined with the language.

6.3.1 Abstract Syntax

The abstract syntax of the property language is shown as a metamodel on Figure 6.4, and is presented thereafter.

6.3.1.1 Propositional variables and structural patterns

A temporal property language must be able to describe *propositional variables* that each represents some truth about the state of the executed model. Such a variable may equal either *true* or *false* for a given state of the model, *i.e.,* it must behave as a predicate

taking the state of the model as parameter. In this chapter, we define such predicates using *structural patterns*. A structural pattern is a partial description of a model that returns *true* if a given model—here, the state of the executed model—matches what it describes, and *false* otherwise. Instead of redefining a structural patterns description language, our temporal property language relies on the existing VQL model querying language. In Figure 6.4, this is represented by the PropositionalVariable metaclass, which has a `name` and an associated VQL Pattern. Such Pattern elements can be defined using the domain concepts of any DSL whose metamodel is imported, by using its metaclasses as Type elements.

6.3.1.2 Scopes

Next, a temporal property language must be able to specify the *scope* of a given property. A scope determines a segment of the execution where a temporal pattern (see below) is expected to be satisfied. On each scope activation, the temporal pattern of the property must be observed, otherwise the property is violated. Instead of reinventing such concept, we adapt the different kinds of scopes used for the PSPs.

In Figure 6.4, this is represented by the Scope abstract metaclass, which is subdivided in different subclasses. The Global scope is active during the whole execution. The Before scope is active until its `upperBound` propositional variable becomes `true`. Conversely, the After scope is active after its `lowerBound` propositional variable becomes `true`, and until the end of the execution. The AfterUntil scope becomes active after its `lowerBound` propositional variable becomes `true` until the end of the execution or its `upperBound` propositional variable becomes `true`, whichever comes first. Finally, the Between works similarly, except that a scope becomes *potentially active* after the `lowerBound` propositional variable becomes `true` and will retro-actively become *active* only once its `upperBound` propositional variable becomes `true`. This means that a verdict cannot be rendered before the scope is confirmed to have been active, and that, as opposed to the AfterUntil scope, the end of the execution is not a valid upper bound for a Between scope.

6.3.1.3 Temporal patterns

Finally, a temporal property language must be able to specify *temporal patterns*, *i.e.,* how propositional variables should take their values when the scope of the corresponding temporal property is active. Like scopes, instead of reinventing temporal patterns, we reuse and adapt the PSPs.

In Figure 6.4, this is represented by the TemporalPattern abstract metaclass, which is subdivided in different subclasses. The Universality (resp. Absence) pattern indicates that its associated propositional variable must be and remain `true` (resp. `false`) during each active scope of the property. The Existence pattern indicates its associated propositional variable must be `true` on a number of execution states that is within the lower and upper bounds of the pattern represented by its `min` and `max` attributes. The Precedence pattern indicates that its associated propositional variable must become `true` before another

```
1  import "http://org.tetrabox.activitydiagram/ad/"
2
3  pattern activeNode(nodeName : java.lang.String) {
4    ActivityNode.name(node, nodeName);
5    check(node.heldTokens.empty() = false);
6  }
7
8  pattern InsertCard() {activeNode("InsertCard");}
9
10 pattern CorrectPIN() {activeNode("CorrectPIN");}
11
12 pattern WrongPIN() {activeNode("WrongPIN");}
13
14 pattern OutputCash() {activeNode("OutputCash");}
15
16 pattern EjectCard() {activeNode("EjectCard");}
```

(a) Structural patterns.

P1	**exists** 1 EjectCard **after** CorrectPIN
P2	EjectCard **precedes** OutputCash **globally**
P3	**exists** [0,2] WrongPIN **between** InsertCard **and** CorrectPIN

(b) Temporal patterns.

Figure 6.5: Example temporal properties for Figure 6.2.

does during each active scope. The Response pattern indicates that during each active scope, when its associated propositional variable becomes `true`, its other propositional variable eventually becomes `true` as well. While the PSPs feature two more patterns, namely the *chained response* and *chained precedence* patterns, we leave them for future work as they are among the least used patterns.

6.3.1.4 Example Properties

To define the temporal properties introduced in Section 6.1, we first need to define the propositional variables corresponding to when specific activity nodes are reached, specifically the `CorrectPIN`, `WrongPIN`, `EjectCard` and `OutputCash` activity nodes. This can be done by defining a general, parameterized structural pattern looking for an ActivityNode whose name is passed as a parameter and whose `heldTokens` property contains at least one Token, as shown on Figure 6.5a. Each propositional variable can then be defined using this general structural pattern.

Once the propositional variables are defined, the temporal properties can be written as shown on Table 6.5b. **P1** is defined as an Existence pattern over an After scope. **P2** is expressed through a Precedence pattern over a Global scope. Finally, **P3** is written as an Existence property with an upper bound, over a Between scope.

6.3.2 Overview of the Translation Scheme

We provide a translational semantics for the proposed temporal property language consisting in separately compiling the structural and temporal patterns of properties into structural and temporal runtime monitors. The resulting monitors are then integrated using a property manager.

6.3.2.1 Structural patterns

As explained in Section 6.3.1, the structural patterns of the proposed temporal property language are directly defined using VQL. Since VQL already has well defined semantics, our translation scheme simply extracts structural patterns from our temporal properties, allowing them to be executed with the semantics of VQL. At runtime, all structural patterns to be monitored are registered as observers of the execution. From then on, the VQL query engine incrementally updates the matches to its registered patterns when modifications are made to the running model. The resulting structural monitors are integrated with the temporal ones by retrieving the current matches of the former and forwarding them to the latter (see below).

6.3.2.2 Temporal patterns

Aside from a small variation on the existence pattern, the intended semantics for the temporal patterns is identical to the existing semantics of the PSPs, which was originally expressed in 3 languages, 2 of which are suitable for runtime monitoring: LTL and Quantified Regular Expressions (QRE).

However, this semantics, expressed as a mapping from pattern/scope combinations to QREs, is defined for finite state verification of infinite traces (*e.g.,* model checking), whereas in our case we seek to provide a semantics for finite execution traces to enable runtime monitoring. While there is existing work providing a semantics to LTL properties on finite traces [BLS11], the ecosystem of tools leveraging this semantics focuses on very specific approaches that do not interface well with our envisioned approach, nor with our technological space. We therefore opted for a different and novel approach: providing QRE-based mappings for the PSPs for finite execution traces, which can then be implemented using widely available CEP frameworks: the event stream is then seen as a string, and QREs define the temporal relationships between event occurrences.

Accordingly, we adapted the existing QRE semantics of the PSPs to the needs of the approach. The result is a translation scheme compiling temporal patterns into CEP-based runtime monitors. Such monitors wait for events from the structural monitors (*e.g.,* matches for the model queries of the propositional values) and eventually deliver their final verdict to the property manager. This substantial adaptation of the original mappings of the PSPs to QREs is detailed in Section 6.4.

6.3.2.3 Integration With the Property Manager

In order to integrate structural and temporal monitors obtained using the translation scheme, we defined a *property manager* that acts as a bridge between the two kinds of monitors. Figure 6.6 illustrates how this works with an example execution of the activity shown in Figure 6.2, where **`P3`** is being monitored. A total of four monitors are deployed in the property manager, three of which are structural monitors evaluating the results of the `InsertCard`, `WrongPIN` and `CorrectPIN` structural patterns. The last monitor is the temporal monitor evaluating the Existence temporal pattern on the Between scope.

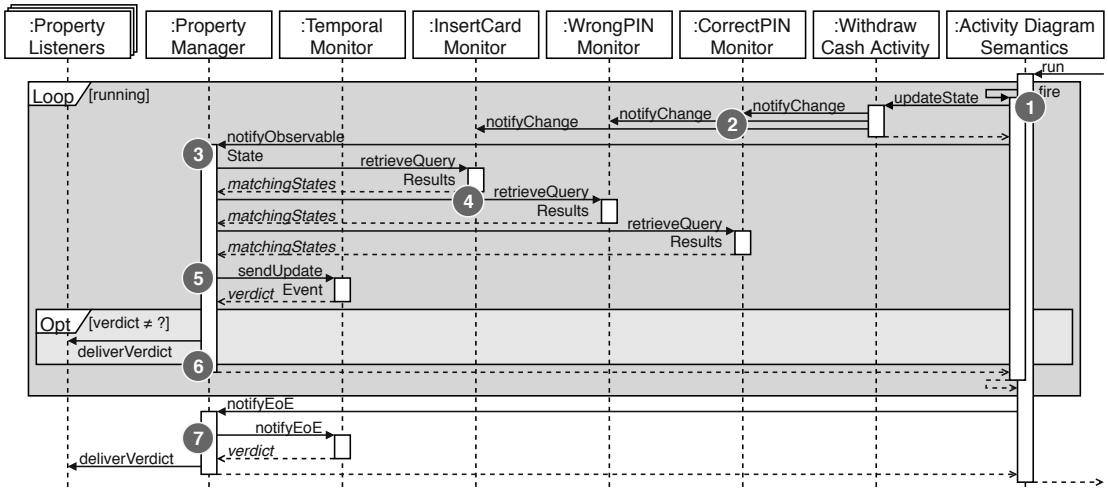


Figure 6.6: Sequence diagram of a monitored execution of the *Withdraw Cash* Activity.

As the execution unfolds, the operational semantics of the *Activity Diagram* DSL repeatedly applies the fire execution rule, which makes changes to the model (**1** on Figure 6.6). Changes are detected by structural monitors, causing them to update their structural pattern result (**2**). Once the execution rule has been applied, the execution semantics notifies the property manager, in conformance with the model execution protocol, that an observable state has been reached (**3**). This causes the property manager to retrieve the current result of each deployed structural monitor (**4**). From these results, the property manager sends an event containing the new values for each propositional variable of the temporal pattern to the temporal monitor. After updating its state, the temporal monitor returns its verdict for the temporal pattern it is monitoring (**5**). If this verdict is a final verdict —*i.e.,* it is either a violation or a satisfaction of the property— then the property manager delivers it to the registered property listeners, and the temporal and structural monitors are unplugged from the execution (**6**). If the end of the execution is reached without a final verdict being delivered, the execution semantics sends the corresponding notification to the property manager (**7**). This notification is forwarded to the temporal monitor, automatically triggering a final verdict which is then delivered to property listeners.

6.4 Translation Scheme of Temporal Patterns for Runtime Verification

In this section, we present the translation scheme used to derive temporal monitors from the temporal constructs of our property language.

6.4.1 Specificities of Runtime Verification

A core difference between our temporal property language and the PSPs is that we evaluate temporal properties at *runtime* and on *finite* executions. Therefore, a final verdict can only be rendered when an execution state that is permanently satisfying or violating the property is reached. In the context of scoped properties, such execution states come in four kinds: (1) states violating the temporal pattern of the property, (2) states satisfying the temporal pattern for the current scope in the case of single scope activation properties (*e.g.,* Globally, Before, etc.), (3) states marking the end of the execution, and (4) states marking the end of a scope activation. Note that a single execution state can belong to the last two kinds, for instance the state marking the end of a Globally scope activation also marks the end of the execution.

While a verdict can always be rendered for the first three kinds of execution states, the fourth kind (states marking the end of a scope activation) does not guarantee this. Reaching the end of a scope activation means that no violation or satisfaction was detected on the basis of a single execution state. This in turn means that a verdict must be rendered based on the states collected by the temporal pattern over the scope activation. One of two outcomes is possible: either a definitive verdict can be rendered (property violated or satisfied), or no definitive verdict can be rendered.

In the first case, a final verdict has been reached, thus the monitoring can be stopped and observers notified of the verdict. In the second case, since the temporal pattern is evaluated independently from one scope activation to another, the resources held about the now inactive scope are not needed to render future verdicts. Therefore, we opted for a monitoring strategy at the scope scale, instead of encompassing the complete execution. As a result, we define the semantics of pattern/scope combinations to encompass individual scope activations instead of the whole execution.

6.4.2 Considered Target Languages

Our semantics relies on identifying matches of temporal patterns among finite sequence of states reached by an executed model, a match being a collection of execution states of interest gathered over a scope activation. Once a match has been found, it is then necessary to analyze the execution states it carries to render a verdict, *i.e.,* whether the temporal pattern is permanently satisfied (noted $\top$), violated (noted $\bot$) or no definitive verdict can be rendered yet (noted '?'). We therefore need to cover two different concerns: the expression of *what states to match*, and the expression of *verdict procedures* that analyze matches. For our approach, a temporal pattern is translated into two objects: a quantified regular expression to express what states to match, and a decision tree to express the verdict procedure.

6.4.2.1 Quantified regular expressions

From a CEP perspective, matches translate well in terms of *complex events*, where the role of events is taken by propositional variables, each being defined beforehand by a

structural pattern. A match can be computed as a complex event emitted upon the reception of a specific pattern of events, and that carries a set of properties whose values are computed from the received set of events.

CEP engines support multiple languages to define complex events, among which QREs. While QREs are more known for the definition of patterns of characters for string searching, they can also be used to search within a sequence of execution states. For example, given three propositional variables `Q`, `P` and `R`, "`Q ¬[P,R]* P`" will search for a match where `Q` is true in an execution state, followed directly by a possibly empty sequence of states where `P` and `R` are false, followed directly by a state where `P` is true.

As Dwyer *et al.* originally expressed the semantics of the PSPs using QREs, relying on QREs to describe our own temporal monitors makes it easier to reuse and adapt the semantics of Dwyer *et al.*.

6.4.2.2 Decision Trees

Since verdict procedures are sequences of tests made on matches, we express them using decision trees. A *decision tree* is a tree where each internal node represents a test on an analyzed object, each branch represents the outcome of a test, and each leaf node represents an outcome of the procedure. In our case, tests focus on specific propositional variables of interest of the corresponding QRE. We later show such variables of interest of a QRE as underlined.

6.4.3 A Translation Scheme for Temporal Patterns

In this part, we use one example of pattern/scope combination to explain how we adapted the QRE-based semantics of the PSPs to runtime verification. This adaptation consists both in changing the QREs proposed by Dwyer *et al.*, and on supplementing each QRE with a verdict procedure written as a decision tree.

Table 6.1 shows an excerpt of the translation scheme we defined for the temporal patterns of our property language, with 3 rules out of 25 in total. Each line shows a pattern/scope combination to the left, and both the QRE and accompanying verdict procedure resulting from compilation to the right. In this part, we focus on the first row of this table, which corresponds to the Existence/Between combination. The complete translation scheme can be found in A.

6.4.3.1 Example of Adaptation of a QRE

Figure 6.7 illustrates with the Existence/Between combination how we rendered the semantics of the PSPs compatible with runtime verification. The original QRE from Dwyer *et al.* is shown at the top, then each arrow is an adaptation step.

As a first step, we restrict the QRE to a single scope activation. This is done by keeping only the part of the expression that is bound by `Q` and `R` (both included). As a second step, we ensure that the QRE is able to capture violations of the property. In our example, this is achieved by making the `P ¬[R]*` pattern optional, which yields

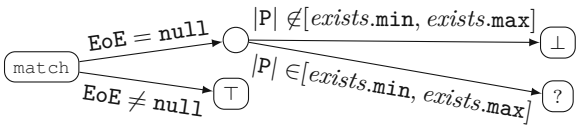
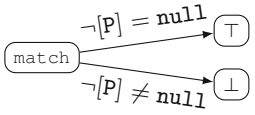
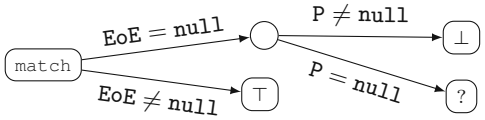
Pattern/scope combinations	QRE semantics and verdict procedure
exists P **between** Q **and** R	EoE | Q ¬[P,R]* (P ¬[P,R]*)* (R | EoE) 
always P **before** Q	P* (EoE | Q | ¬[P]) 
S **precedes** P **after** Q **until** R	EoE | Q ¬[EoE,P,R,S]* (EoE | P | R | S) 

Table 6.1: Excerpt of Pattern/Scope combinations and their corresponding semantics as QREs and verdict procedures.

exists P **between** Q **and** R:

(¬[Q]* Q ¬[P,R]* P ¬[R]* R)* ¬[Q]* (Q ¬[R]*)?

⇓ **1**

Q ¬[P,R]* P ¬[R]* R

⇓ **2**

Q ¬[P,R]* (P ¬[R]*)? R

⇓ **3**

Q ¬[P,R]* (P ¬[P,R]*)* R

⇓ **4**

EoE | Q ¬[P,R]* (P ¬[P,R]*)* (R | EoE)

Figure 6.7: Example of step-by-step adaptation of the semantics from Dwyer *et al.* [DAC98] to runtime verification.

(P ¬[R]*)?. This way, the QRE can capture scope activations where P does not occur. The third step is specific to the Existence temporal pattern: we want this pattern to gather all occurrences of P happening while the scope is active. This allows us to support lower- and upper-bounded Existence patterns by comparing the number of occurrences of P contained by a match to the min and max properties of the temporal pattern. This is achieved by changing the (P ¬[R]*)? sub-expression from the previous step into (P ¬[P,R]*)*. This way, additional occurrences of P that would have been captured by ¬[R]* instead lead the (P ¬[P,R]*) pattern to be matched repeatedly. The fourth

and final step consists in introducing an EoE (end of execution) event into the QRE. This event must first be added as an alternative to the whole pattern, so that when no scope is currently active, a complex event is still generated, triggering the rendering of a verdict. The second place where EoE must be added is as an alternative to R, in order to trigger the verdict rendering during a potentially active scope. To obtain our complete semantics, we repeated these adaptation steps for each of the QREs part of the semantics from Dwyer *et al.* [DAC98].

6.4.3.2 Example of Verdict Procedure

As we previously explained, for the purpose of runtime verification, each QRE requires a companion verdict procedure to compute an outcome from a sequence of matched execution states. A verdict procedure is executed each time its QRE finds a match, and delivers a verdict based on the properties of the complex event representing the match. These properties contain the values of a set of propositional variables of interest.

In what follows, we focus on the first row of Table 6.1, which corresponds to the same Existence/Between example, whose variables of interest are P and EoE (shown as underlined). The verdict procedure first checks if it was triggered by the end of the execution. If that is the case, as evidenced by the EoE event not being null, the temporal property is satisfied. This is because the Between scope does not consider the end of the execution as a valid scope upper bound, contrary to the AfterUntil scope. If the end of the execution was not reached, this means that the verdict rendition was triggered by a R event. In this case, the procedure counts the number of occurrences of P and checks that it is within the lower and upper bounds of the Existence temporal pattern. If the number of occurrences of P stays within these bounds, the property is neither satisfied nor violated. Otherwise, the property has been violated, and a verdict is rendered accordingly. To obtain our complete semantics, we wrote a verdict procedure for each QRE defined.

6.4.3.3 Example of an Execution of a Temporal Monitor.

Figure 6.8 shows the evaluation of **P3** on an example execution of the *Withdraw Cash* activity shown in Figure 6.2. The upper part of the figure shows an excerpt of the execution trace resulting from the execution. Here, an incorrect PIN is entered the first time, eventually followed by the correct PIN, thereby constituting a full scope activation for **P3**, as depicted between execution states **1** and **5** on the upper left part of Figure 6.8. Once execution state **5** is reached, the QRE semantics of the Existence/Between combination captured a full match, as shown on the middle left part of the figure. This match in turn triggers the associated verdict procedure, shown on the lower left part of the figure. As the end of the execution has yet to be reached and the number of WrongPIN occurrences is within the bounds of the Existence pattern, no final verdict is produced ('?').

After the verdict is rendered, the execution resumes, as shown on the upper right part of Figure 6.8. For this execution, no additional scope activation are encountered

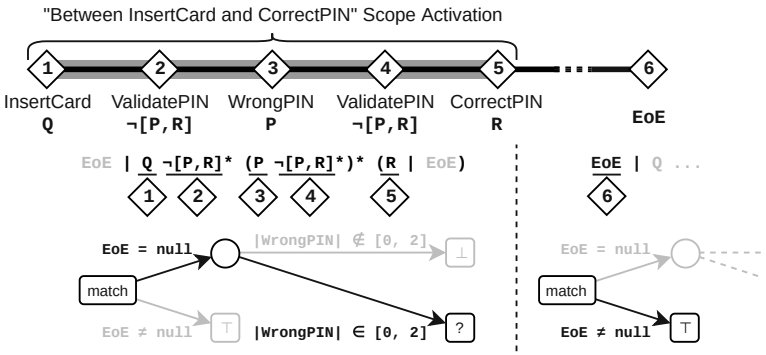


Figure 6.8: Evaluation of **P3** on an execution of the *Withdraw Cash* activity from Figure 6.2.

until the end of the execution. When the end of the execution is reached, the temporal monitor receives a corresponding notification. As the QRE of the temporal monitor is not currently matching anything, it directly captures the EoE occurrence, thereby completing a match as shown on the middle right part of the figure. This match triggers the execution of the verdict procedure, which this time returns a satisfied verdict ($\top$), as shown on the lower right part of the figure.

6.5 Evaluation

In this section, we leverage our implementation of the approach to evaluate the execution overhead induced by monitoring temporal properties on a selection of xDSLs. We refer the reader to Section 7.4 for more details on how the approach was implemented. We seek to investigate how different factors influence this overhead and seek to answer the following research questions:

RQ#1 How does each property influence the execution overhead?

RQ#2 How does model size influence the execution overhead?

RQ#3 How does the footprint of properties influence the execution overhead?

RQ#4 How does execution length influence the execution overhead?

The evaluation material (models, code, data) is available on the companion web page[1].

Considered xDSLs For this evaluation, we considered three xDSLs: an extension of the *Activity Diagram* DSL presented in Section 6.1, *MiniJava* [Rob01] and *ThingML*[2]. Our *Activity Diagram* implementation was initially proposed as a solution for the Model Execution case of the Transformation Tool Contest 2015 [CDB+14]. In addition to what

[1] http://gemoc.org/ecmfa20/

[2] https://github.com/TelluIoT/ThingML

is shown in Figure 6.1, it comprises the concepts of Variable and Expression. Global variables can be declared in an activity, and the values of these variables can be changed by action nodes using integer or boolean expressions. Guards can also be used in the control flow using boolean expressions. *MiniJava* is a subset of Java created for teaching purposes. *ThingML* is a DSL for designing and implementing distributed reactive systems. It combines asynchronously communicating statecharts and components, an imperative platform-indenpendent action language and constructs targeting IoT applications. All three DSLs were implemented with the GEMOC Studio using Ecore for the abstract syntax and Kermeta for the operational semantics.

Considered Models To evaluate how the approach scales with regards to different factors, we generated models covering two factors: *model size* and *execution length*. This allows to evaluate the impact on the induced overhead of an increased search space and execution length. Models for *Activity Diagram* and *MiniJava* are generated from the same template: a loop performing calculations on integer variables for a number of iterations. For these DSLs, we modulate model size by replicating both the integer variables that are part of the structural patterns used by the evaluated temporal properties, and the calculations made on these variables. We cover models containing around 15, 150 and 1500 integer variables, referred to as factors *S*, *M* and *L*, respectively. To modulate execution length, we increase the number of executed iterations. We cover models iterating 10 and 100 times, thereafter referred to as *short* and *long* execution lengths, respectively. Models for *ThingML* consist of clients sending registration notices and signal to servers. When servers receive a signal from all of their registered clients, they send them a signal in response. Clients can switch to another server or shutdown depending on the total number of received signals. For *ThingML*, we modulate model size by adding more clients to the system, and we modulate execution time by changing the number of signals sent by clients. We cover models containing 5, 25 and 75 (resp. S, M, and L) each sending 3 to 20 (resp. short and long execution lengths) signals to the servers.

Considered Temporal Properties In order to evaluate how the approach scales with regard to *property footprint*, we generated temporal properties whose structural patterns cover portions of the models of various sizes. For *Activity Diagram* and *MiniJava*, we consider temporal properties that focus on the variables of the models. These variables are designed so that each structural pattern featured in temporal properties has its corresponding variables. For instance, the structural pattern tied to the `Existence` temporal pattern searches for a variable named `"existVar_0"`. The footprint of `Existence` properties is multiplied by 10 by generating structural patterns that require matches on `existVar_0` through `existVar_9`. For these 2 DSLs, we cover structural patterns matching 1, 10 and 100 variables, thereafter referred to as *S*, *M* and *L*, respectively. In the case of *ThingML*, temporal properties are based on patterns that check the current state of specific clients. For example, one type of client is expected to receive 5 events between 2 register notices, so we can check that the state `Waiting`, reached after receiving a signal, exists 5 times between the states `Registered` and `Register`. For this DSL, we

Execution length	Short						Long						mean
Model size	S	M		L			S	M		L			
Property footprint	S	S	M	S	M	L	S	S	M	S	M	L	
Baseline execution time (in s) / Mean execution time (in s) / *Mean relative execution overhead* (in %)													
MiniJava	0.05	0.21		1.44			0.25	1.15		11.98			0.61
	0.11	0.27	0.31	1.77	1.84	12.03	0.34	1.87	1.89	16.22	15.72	26.98	1.95
	134.5	*23.5*	*45.0*	*21.1*	*24.9*	*725.8*	*38.7*	*63.0*	*63.7*	*34.7*	*30.1*	*122.4*	*56.5*
ThingML	0.10	0.73		5.10			0.31	2.17		15.35			1.27
	0.19	1.20	1.25	9.30	9.30	9.17	0.49	3.66	3.62	24.49	24.02	24.03	4.08
	77.1	*62.6*	*69.4*	*82.0*	*82.0*	*79.3*	*59.3*	*68.8*	*66.8*	*59.3*	*56.2*	*56.4*	*67.6*
Activity Diagram	0.03	0.05		0.08			0.17	0.19		0.60			0.12
	0.09	0.11	0.16	0.23	0.26	10.36	0.24	0.35	0.40	1.52	1.60	12.06	0.56
	168.7	*120.4*	*230.5*	*194.3*	*234.9*	*13053.6*	*40.2*	*83.2*	*105.0*	*150.9*	*164.2*	*1891.0*	*244.6*
All 3 DSLs	*120.1*						*79.6*						*97.8*

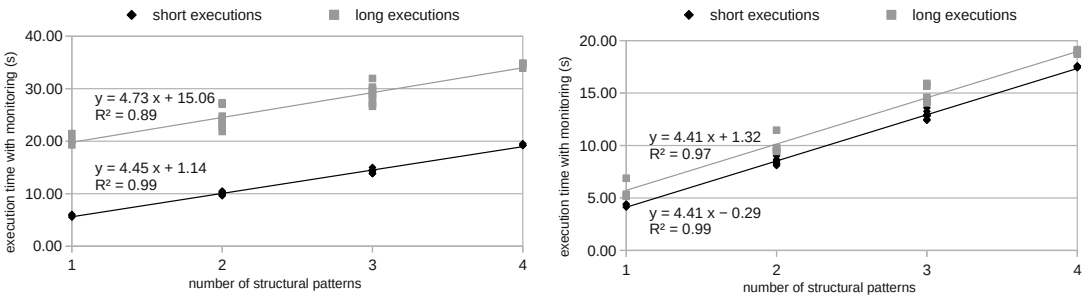
Table 6.2: Average execution overhead for each factor (using geometric mean, all properties combined, S=small, M=medium, L=large).

cover structural patterns matching 1 (S), 5 (M) and 10 clients (L). Finally, all temporal properties are designed to be validated at the end of the execution only, to measure their overhead during the complete execution.

Experimental Protocol The execution overhead of each combination of temporal property, model size, execution time and property footprint is measured as follows. First, a warm-up phase consisting of 10 executions of the model while monitoring the property takes place. We then collect the execution time of 20 additional executions, still while monitoring the property, and compute the average execution time. We do the same for the mean base execution time of the model, without monitoring. Summaries of the resulting measurements are presented in Table 6.2.

Analysis Table 6.2 shows, for each combination of factors and for each DSL, the base execution times of the models, the (geometric) mean execution times while monitoring across all properties, and the resulting relative execution overheads.

The results show that the average overhead ranges from 56% (for *MiniJava*) to 244% (for *Activity Diagram*). A more detailed look shows that, for *Activity Diagram*, runtime monitoring on short executions induces a very high (and even extremely high, in the case of the highest property footprint) overhead: from 168% to 13053% in the most extreme case, whereas the overhead on longer executions is more reasonable on property footprint S (40% to 150%) and M (105% to 164%), but still very high on property footprint L, with 1891% execution overhead. In comparison, the overhead measured for *ThingML* stays within a reasonable range for all factors (from 56% to 82%), whereas in the case of *MiniJava*, both model sizes S and L show, respectively, high (134%) to very high



(a) MiniJava (b) Activity Diagram

Figure 6.9: Execution time per number of structural pattern in properties (model size L, property footprint L).

(725%) overhead on the shortest execution time. Using Table 6.2, we answer each of the considered research questions in the following.

Answering RQ#1 From Table 6.2, one can observe that for both execution lengths, the execution times while monitoring properties with footprint L are between 10s and 15s higher (*i.e.,* one order of magnitude higher) than the baseline execution time for *MiniJava* and *Activity Diagram*, while staying closer to the baseline (*i.e.,* within the same order of magnitude) for footprint S and M. The same discrepancy between footprint L and S/M cannot be observed for *ThingML*.

Looking at the detailed data (available in Appendix B) reveals that the overhead induced by property footprint L for *MiniJava* and *Activity Diagram* is highly dependent on the number of structural patterns contained in the properties. This is illustrated by Figures 6.9a and 6.9b, which provide a more detailed look at these two outliers and show that there is a highly-correlated (with R^2 going from 0,89 to 0,99) relationship between the number of structural patterns contained in the properties and the resulting overhead. This relationship estimates the absolute induced overhead per structural pattern to be situated between 4,41s and 4,73s.

We thus explain the difference between the 3 DSLs as follows: there is a steep initial overhead depending on the amount and size of structural patterns on a property, that gets compensated by longer execution times. For *ThingML*, the induced overhead is reasonable because, on one hand structural patterns for our *ThingML* models are smaller by an order of magnitude and thus their initialization has less impact on the induced overhead, and on the other hand, *ThingML* is the slowest of all 3 languages, as evidenced by the baseline execution times shown in Table 6.2.

In the end, the most salient source of overhead from the kind of property being monitored comes from the *number of structural patterns* a property contains. This source of overhead appears to be confined to the *initialization phase* of structural patterns, and therefore is smoothed out on longer executions. Identifying the influence of each pattern and scope making up temporal properties requires a more in-depth statistical analysis focused solely on this aspect, which we reserve for future work.

Answering RQ#2 For *MiniJava*, increases in model size greatly increase the

execution time, resulting in a reduced relative overhead: without monitoring any property, going from model size S to L results in around 28 times longer executions. For *ThingML*, increases in model size have even more impact on the execution time (going from size S to L results in executions around 51 times longer), further compensating the initial overhead. Finally, for *Activity Diagram*, increases in model size do not sufficiently increase the execution time to have noteworthy compensating effects.

From these observation, and especially by comparing the respective overheads of the 3 DSLs, it appears that increase in model size has a relatively small influence on the induced overhead due to the search space increase for structural patterns, which in the case of *MiniJava* and *ThingML* is compensated by the increased execution time resulting from increased model size. In future work, we plan to further investigate if pattern search space is tied to structural pattern initialization time.

Answering RQ#3 Due to the steep initial overhead from structural patterns identified above, increasing property footprint by an order of magnitude can have a drastic effect on the induced overhead, especially on the shorter execution times. This initial influence of property footprint is compensated on longer execution time. Due to the high initial overhead induced by structural patterns, our data does not allow us to conclude on the role of increased model footprint on the overhead, past the initial one.

Answering RQ#4 Finally, the results show an average overhead of 120% on short executions, and of 79% on longer executions. Looking at the detailed data also shows that longer executions smooth out the difference of overhead between temporal properties. This again hints at an absolute overhead induced by using the approach at all, both compensated and smoothed out across the properties on longer executions.

Summary We conclude that the approach is well-suited for testing and interactive debugging, where an average execution overhead of 79% for middle length executions, and of 120% for very short executions are reasonable. While this seems high compared to existing language-specific approaches, we argue that state-of-the-art approaches rely on language-specific, low-level optimizations whose reproduction in a generic approach raises new scientific challenges. For scenarios such as live analytics, an asynchronous variant of the approach would be more suitable, and is left for future work.

6.6 Summary

Online behavioral analysis is required for reactive models to provide precise control over when event occurrences are sent to running models. To enable this, the main challenge at hand is to allow domain experts to express their temporal properties at the domain level, while providing a language-agnostic way to evaluate these properties. To this end, we proposed a temporal property language based on the high-level and time-honored concepts of temporal patterns and scopes concerning the temporal part of the properties, and on domain-level queries concerning their structural part. We complement this temporal property language with a runtime monitoring backend based on the model execution protocol, allowing to deploy the properties as runtime monitors and evaluate them during the execution of models. We implemented the temporal property language

and the runtime monitoring backend as part of the GEMOC Studio, and we performed a quantitative evaluation of the performances of our implementation. These measurements indicate that the performances of the approach are within an acceptable range for the envisioned use cases of testing and debugging.

In the next chapter, we provide more details on the different artifacts resulting from the implementation of our contributions as part of the GEMOC Studio.

CHAPTER 7

Tool Support in the Context of the GEMOC Studio

In the three previous chapters, we contributed to two distinct yet complementary research axis. The first axis is dedicated to the extension of the scope of both the xDSLs supported by existing generic tools and the generic tools that can be defined for xDSLs. We contributed to this axis by providing a generic approach for the definition and exposition of the behavioral type of xDSLs. The second axis is dedicated to the provision of generic behavioral analysis facilities for xDSLs. Our contribution on this axis is split between offline and online facilities: the trace comprehension operators provide offline facilities, while the temporal property language, coupled with the runtime monitoring backend, provide online facilities.

In this chapter, we first present the GEMOC Studio [BDV+16] in Section 7.1 whose model execution protocol —implemented by its execution engines— is the cornerstone of (*i*) the metalanguage integration facade of the event manager, (*ii*) the (preexisting) generic execution trace recorder on top of which our trace comprehension operators are defined, and (*iii*) the runtime monitoring backend. In Section 7.2, we present the event management facilities resulting from our work on behavioral language interfaces for xDSLs presented in Chapter 4. We then detail in Section 7.3 how we implemented the trace comprehension operators presented in Chapter 5, and the tools leveraging them to provide advanced trace analysis facilities. Next, in Section 7.4 we provide more detail on the inner workings of the translation scheme from our temporal property language to our runtime monitoring backend, both proposed in Chapter 6. Finally, in Section 7.5 we propose an envisioned integration of all these artifacts as an extended test runner leveraging all the contributions in one tool.

The code of the implementation of our contributions presented in this chapter is open-source and available on Github [1].

[1] `https://github.com/eclipse/gemoc-studio-modeldebugging`

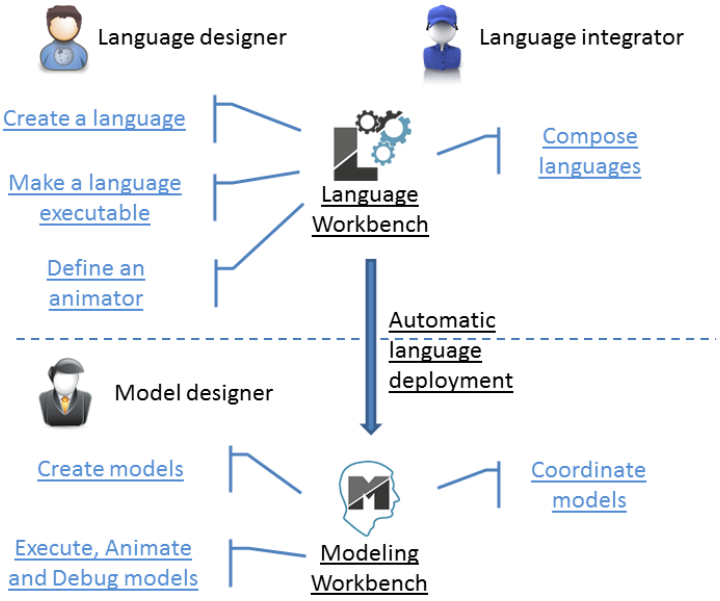


Figure 7.1: Roles and services provided by the GEMOC Studio.

7.1 Presentation of the GEMOC Studio

The GEMOC Studio[2] is an Eclipse package atop the Eclipse Modeling Framework (EMF) [SBMP08]. Figure 7.1 shows an overview of the roles and services related to the two workbenches composing it, namely the language workbench on the upper part, and the modeling workbench on the lower part. The language workbench is used by language engineers to define new xDSLs and their tools. This includes defining the abstract syntax (using Ecore), the operational semantics (using Kermeta [JCB+13], MoCCML, ALE, xMOF [MLWK13] or Henshin [ABJ+10]) and the concrete syntax (using Sirius Animator or Xtext) of the language. In addition, the GEMOC Initiative aims at enabling the composition of xDSLs by defining how the execution of models can be coordinated.

Then, xDSLs defined in the language workbench are automatically deployed into the modeling workbench, which provides an IDE for the edition, execution, animation and debugging of conforming models. The modeling workbench provides an execution engine for each of the aforementioned metalanguages, allowing to execute any model conforming to an xDSL defined within the language workbench. The modeling workbench allows executed models to be animated, provided they have a graphical syntax and the language engineer defined how to represent dynamic data during the execution. The

[2] http://gemoc.org/studio.html

modeling workbench also provides generic tracing facilities for executed models, as well as a model debugger that can be used on models conforming to any xDSL defined within the language workbench, which comes with an extension for omniscient debugging that relies on the execution trace of the model. At the core of these functionalities is the execution framework of the GEMOC Studio.

This execution framework provides unified interfaces for execution engines and execution listeners. The model execution protocol presented in Section 3.3 relies on these two interfaces: execution engines for specific metalanguages all implement the common execution engine interface, while runtime tools implement the common execution listener interface. This allows to decouple the tools from the execution engine used to run the execution, and thus from the metalanguages used to define xDSLs. During the execution, an execution engine sends notifications to all its execution listeners on various execution-related events, such as when the execution starts, before and after calls to execution rules, or when the execution stops. This informs execution listeners that a consistent execution state has been reached. They can thus query the model state to update a view or to log information. As these notifications are synchronous, execution listeners can even control the execution of the model when they handle a notification, for instance by pausing it or calling an execution rule. This mechanism is used for both implementing the generic omniscient debugger that comes with the GEMOC Studio, and realizing the metalanguage integration facade for the event manager proposed in Section 4.4.2.

7.2 Event Management Facilities

The provided event management facilities involve all possible user roles in the MDE process. We thereafter describe what are the necessary steps (if any) that each role must perform, and what kind of support we provide out-of-the-box for applying or using the approach as a user adopting a given role.

Facilities for Metalanguage Engineers As described in Section 4.4.2, we rely on metalanguage integration facades to keep the event manager agnostic of the metalanguage used. Each facade is defined for a specific metalanguage (and thus provided by metalanguage engineers), and implements the execution listener interface. An integration facade must provide the means to call arbitrary rules from the execution semantics of the xDSLs defined with the metalanguage to which the integration facade is dedicated. In the scope of this thesis, we implemented an integration facade for the Kermeta metalanguage. During the initialization phase of an execution, this integration facade loads the specification of the DSL and recovers the set of execution rules from the operational semantics. Then, during the execution, it handles call requests through reflective calls to these execution rules, performed with the `java.lang.reflect.Method.invoke` method.

Facilities for Language Engineers To facilitate the definition of new behavioral interfaces by language engineers, the metamodel of the behavioral interface language (defined in Ecore, and detailed in Section 4.3.1) is complemented by a textual concrete

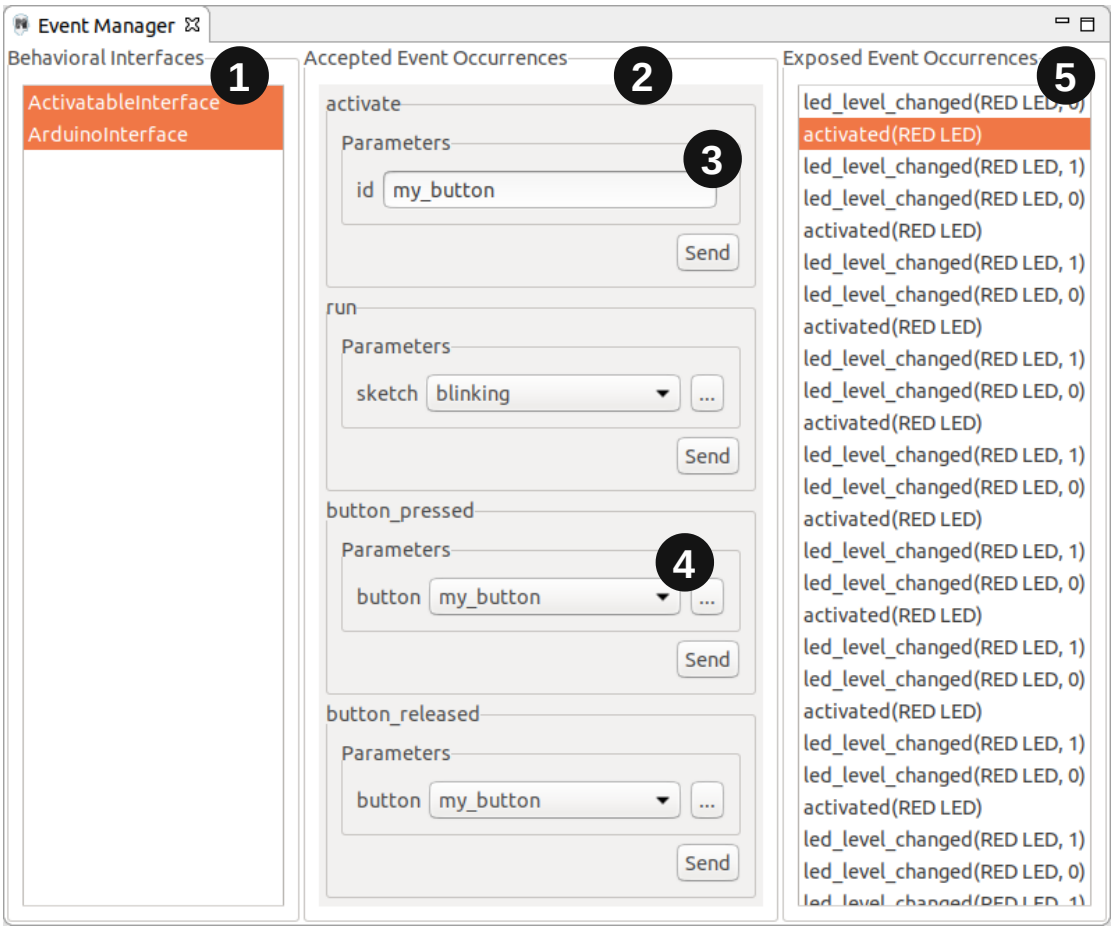


Figure 7.2: Reflective event injection GUI.

syntax (defined in Xtext). In addition, we provide a library for defining implementation and subtyping relationships. This library provides abstract relationship classes following the template method design pattern. Defining a relationship this way requires the language engineer to specify each part of the ECA rules constituting the relationship: the event part is defined as an EPL statement, and the condition and action parts are defined by implementing their respective methods, returning a boolean for the condition and a sequence of event occurrences or call requests for the action. This library also provide simpler implementations relationships —mapping events to similarly named execution rules— that only need to be configured by providing a behavioral interface, a set of execution rules, and an optional mapping between event names and the name of their corresponding execution rule (a default mapping is used otherwise).

Facilities for Modelers Finally, as mentioned in Section 4.5, we designed a reflective event injection GUI for modelers to be able to manually use the approach, which is shown

```
interface VirtualTrace extends MultidimensionalTrace {
  static VirtualTrace load(MultidimensionalTrace originalTrace) {...}

  MultidimensionalTrace export();

  VirtualTrace removeDimension(Dimension dimensionToRemove);

  VirtualTrace reduce();

  TraceComparisonResult compare(VirtualTrace otherTrace);

  TraceComparisonResult compare(VirtualTrace otherTrace,
      Map<Dimension,Dimension> dimensionMapping);

  StateGraph stateGraph();
}
```

Listing 7.1: The `VirtualTrace` API.

in Figure 7.2. On the left (labeled **1** on the figure) are listed all implemented and supertype interfaces. By selecting one or several behavioral interfaces in this list, the middle section (labeled **2** on the figure) is updated to provide one event occurrence configurator per accepted event defined in the selected interfaces. By reflectively analyzing the defined parameters for each accepted event, the GUI is able to provide well-suited controls to configure an occurrence of these events. For example, in Figure 7.2, the configurator for `activate` event occurrences (labeled **3**) provides a text field that lets users enter the value of their choosing for the `id` parameter, whose type is a string. Alternatively, the configurator for `button_pressed` event occurrences (labeled **4**) provides a list of all model elements whose type matches the parameter type (here PushButton), as well as a browse button that lets users select a predefined model element in an arbitrary resource located in the workspace. Finally, the right part of the GUI (labeled **5** in the figure) consists of a log of exposed event occurrences listing all the received event occurrences in reverse chronological order.

7.3 Execution Trace Analysis Facilities

At the core of our implementation of the execution trace analysis facilities is a backend implemented as a fluent API for the creation, manipulation, and analysis of *virtual traces*. Atop this backend, we built a set of graphical views to display execution traces in a human-readable way and to apply the various comprehension operators on them. Hereafter, we provide more details on the API, and then cover the graphical views.

7.3.1 The VirtualTrace API

Listing 7.1 shows the VirtualTrace API regrouping the four trace comprehension operators. Virtual traces defined in this API actually act as a wrapper for the multidimensional traces used in the GEMOC Studio, which conform to the corresponding metamodel proposed in

[BLC+18]. As such, they provide, in addition to the methods shown in Listing 7.1, a set of accessors to query the values of particular execution states or dimensions. In addition, virtual traces only contain links to the concrete input trace instead of containing values (hence their name), along with information on the filtered dimensions and the regrouped states. This limits the amount of memory required to manipulate the trace by avoiding the duplication of the values contained therein.

The first two methods of the API are dedicated to the loading and exporting of execution traces. Once a virtual trace is created through the `load` method, the two trace manipulation operators can be used. Then, a virtual trace can be exported as a new, self-contained multidimensional trace through the `export` method, possibly to be serialized. The next two methods, `removeDimension` and `reduce`, correspond to the *Filter* and *Reduce* operator, respectively. The `removeDimension` method takes a dimension as a parameter and returns a virtual trace where this dimension has been filtered out. The `reduce` method performs a reduction of the virtual trace by grouping together equivalent successive states with regard to the active dimensions. Using these first methods, one can thus write statements chaining the various operators, such as:

```
final MultidimensionalTrace newTrace = VirtualTrace.load(myTrace)
        .removeDimension(someDimension)
        .removeDimension(someOtherDimension)
        .reduce()
        .export();
```

Next, the API offers two `compare` methods, implementing the *Compare* operator. The first method takes as parameter the virtual trace to compare the current trace to, and a map of dimensions. The second method additionally takes a map from dimensions of the current trace to dimensions of the other trace, allowing to specify a custom mapping between the dimensions of each trace. In the default case (*i.e.,* when using the first method or when supplying an empty map to the second method), the dimensions are assumed to be the same. Our implementation of `compare` relies on the algorithm to compute the Levenshtein distance between strings[Lev66], which we adapted to operate on multidimensional traces. The method produces a serializable alignment model, instance of the TraceComparisonResult class. Note that this output model is not self-contained as it references execution states of the compared traces.

Finally, the API offers the `stateGraph` method, implementing the *Graph* operator. This method produces a serializable state graph model, instance of the StateGraph class, according to the specification of the operator given in Section 5.3.5. As with the trace alignment model, this state graph model is not self-contained and references execution states from the input trace.

7.3.2 Trace Visualization Facilities

On top of the VirtualTrace API, we provide a set of graphical views allowing modelers to use the operators through the graphical user interface. Figure 7.3 shows these views on an execution of a state machine model, whose AST is shown on the left.

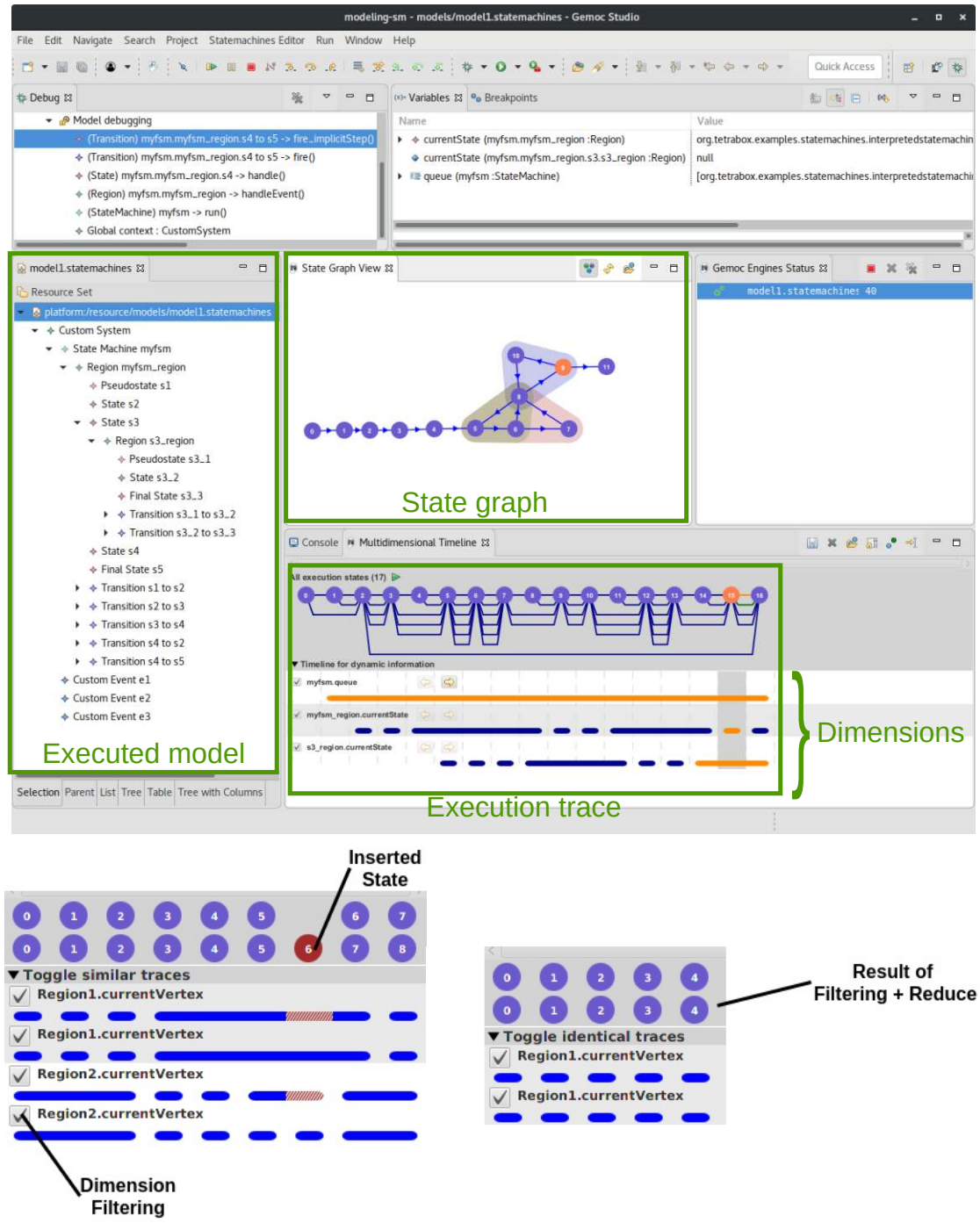


Figure 7.3: Graphical views built on top of the trace comprehension operators.

At the center of the figure, the visualization of the multidimensional execution trace is shown. This graphical view is constituted of a sequence of execution states at the top, linked together by the execution steps leading from one state to another, and of the individual dimensions at the bottom, each containing the trace of its values. The active execution state and values are highlighted in orange. Each dimension can be filtered out or added back in by checking off or on its corresponding checkbox. In this visualization, the *Reduce* operator is applied systematically after a dimension is added or removed.

On top of the trace visualization, the view corresponding to the state graph extraction operator is shown. This view allows to load an existing trace but can also display the state graph corresponding to the execution trace of the running model, as is the case in the figure. When that is the case, the current state of the model is highlighted in the same way as for the trace visualization. As an additional functionality, this view offers the optional highlighting of all the cycles in the graph, as shown on Figure 7.3.

Finally, at the bottom of the figure, the graphical view used for trace comparison is shown. This view is not synchronized with the ongoing model executions. By loading two execution traces, users can obtain a visualization of the comparison result between the traces that is similar to the visualization use for single execution traces: the sequences of execution states and the pairs of equivalent dimensions are displayed on top of one another, allowing an easy comparison. The differences between equivalent dimensions and the inserted, removed or substituted execution states are all highlighted in red. This view also allows to filter out individual dimensions from either execution trace, through the same mechanism of checkboxes. When a dimension is filtered out or added back in, the result of the comparison is recomputed and the visualization is refreshed. Figure 7.3 shows how filtering out the `Region2.currentVertex` dimension produces a perfect alignment of the traces.

7.4 Property Monitoring

We implemented the metamodel of the temporal property language using Ecore, on top of which we defined a textual concrete syntax (with Xtext) allowing to import and reference structural patterns defined in VQL.

The semantics attributed to each pattern/scope combination through our translation scheme is defined using a utility library provided as part of our runtime monitoring backend. This library provides a RuntimeMonitor class, to be configured with an EPL statement feeding a stream of complex events, and with a Java method called on each such complex event added to the stream and returning a truth value (true, false or unknown) for the property being monitored. In the case of our translation scheme, the EPL statement provided for each pattern/scope combination follows their respective QRE semantics, as detailed in Section 6.4, and the decision tree constituting the verdict procedure is encoded in the aforementioned Java method. Listing 7.2 shows how the semantics of the Always/Before combination shown in Table 6.1 is implemented using the runtime monitoring backend.

```
class AlwaysBefore implements TemporalPropertySpecification {
  private final IQuerySpecification<?> p;
  private final IQuerySpecification<?> q;
  private final String name;

  public AlwaysBefore(Pattern always, Pattern before) {
    p = always;
    q = before;
    name = "Always_" + p.fqn + "_Before_" + q.fqn;
  }

  @Override
  public String getStatementString() {
    final String result =
      "select * from" + name + "\n" +
      "match_recognize (" + "\n" +
        "measures nP as nP" + "\n" +
        "pattern (P* (EoE | Q | nP))" + "\n" +
        "define" + "\n" +
        "P as P." + p.fqn + "? is not null," + "\n" +
        "nP as nP." + p.fqn + "? is null," + "\n" +
        "Q as Q." + q.fqn + "? is not null," + "\n" +
        "EoE as EoE.executionAboutToStop? is not null" + "\n" +
      ")";
    return result;
  }

  @Override
  public TruthValue getVerdict(Map<String, List<Map<?, ?>>> events) {
    List<Map<?, ?>> lnP = events.get("nP");
    return if (lnP == null || lnP.empty) TruthValue.SATISFIED else TruthValue.VIOLATED;
  }
}
```

Listing 7.2: Implementation of the Always/Before combination.

The `getStatementString` method is used to provide the appropriate QRE as an EPL statement. This statement first declares an event stream named after the name of the property being monitored. Note that, in our implementation, the events in this stream are in fact maps associating, to each structural pattern, the list of all their matches in the running model. Next, the statement specifies the content of the complex events it will produce when finding matches for its pattern: here, resulting events will only contain the event matched for the `nP` symbol, stored under a property named `"nP"`. We then provide the QRE constituting the pattern of the statement, which corresponds to the one given in Table 6.1. Finally, each symbol used in this QRE defines a subset of the events composing the input stream: `P` and `Q` both include input events holding a property named after the fully qualified name of their respective structural pattern, while `nP` explicitly includes events that do not hold a property named after pattern `p`, and `EoE` matches events containing an `executionAboutToStop` property signaling the end of the execution. Note that, in the case of alternations such as `EoE | Q | nP`, the event processing engine will try to match input events to these symbols in the order they appear. This means that, while the definition of these three symbols are not exclusive,

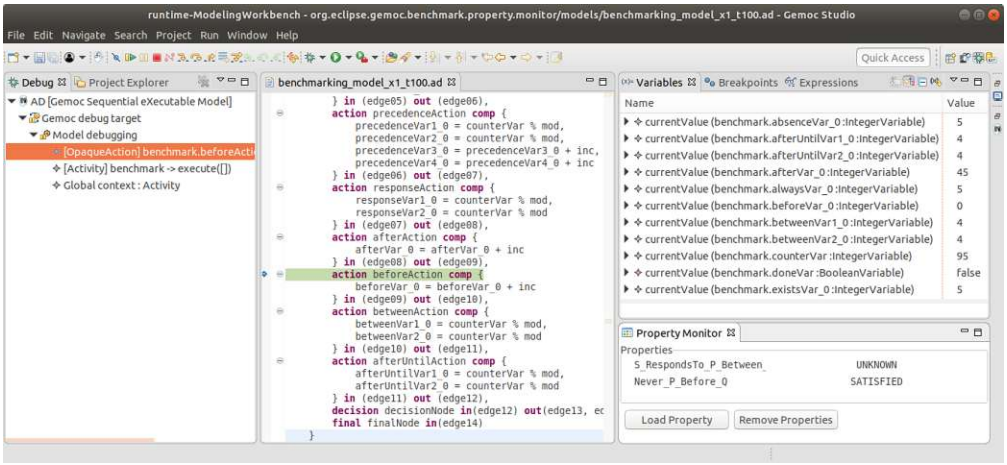


Figure 7.4: Screenshot of the modeling workbench while monitoring properties on an Activity Diagram model.

an event matching `EoE` or `Q` will never be matched as `nP` for instance.

The verdict procedure of the Always/Before combination is then implemented as the `getVerdict` method. This method is called when a complete match is found for the pattern defined as part of the EPL statement, and is passed the event resulting from this match as a parameter. In the example, the method is thus always called with an event containing a property named `nP`. For this particular combination, the temporal property is only validated if the `nP` property (corresponding to `¬[P]` from Table 6.1) is `null` or empty. The temporal property is otherwise violated. Instances of RuntimeMonitor can then be deployed into the property manager, which doubles has an execution listener, to be evaluated at runtime.

Runtime Monitoring for Modelers To allow modelers to easily load their temporal properties as runtime monitors, and to provide real-time feedback on the verdict of each registered property during an execution, we provide a GUI listing the properties along with their verdict among `validated`, `violated`, or `unknown`. Figure 7.4 shows this GUI, in the lower right corner, being used to monitor properties in the modeling workbench of the GEMOC Studio.

Defining Temporal Properties from Execution Traces An envisioned interaction between the *Filter* and *Reduce* operators and the definition of temporal properties is the direct generation of the VQL patterns serving as structural patterns. For a given model state, for each of its considered dimensions, a VQL pattern can be generated, representing the corresponding subgraph of the model state. A step further is the provision of a set of graphical operations allowing to define a temporal property by selecting, directly on the trace visualization, the execution states from which to generate the structural patterns for the scope and temporal pattern of the property.

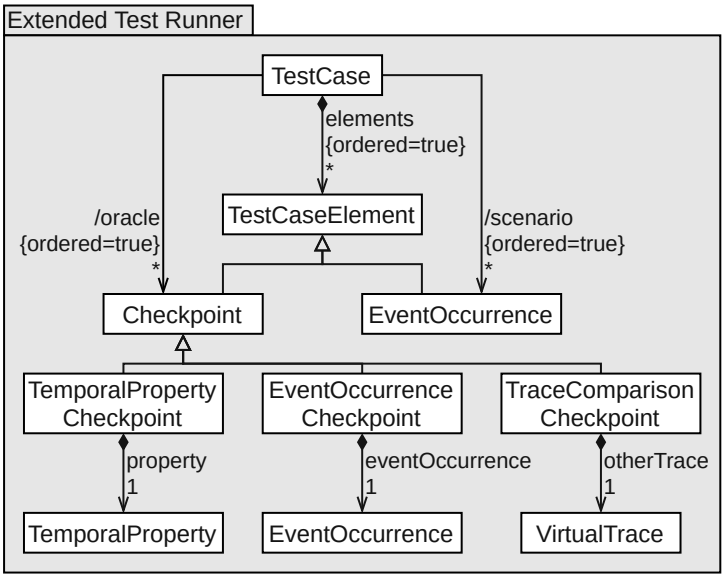


Figure 7.5: Extended test case metamodel.

7.5 Extended Test Runner

As a way to evaluate our contribution on the behavioral typing of xDSLs, we designed a test runner (see Section **Req. 2**) able to process test cases constituted of a test scenario (*i.e.,* a sequence of event occurrences to be sent to the model under test) and a test oracle (*i.e.,* a sequence of event occurrence to be received from the model under test). Based on the additional contributions regarding the online and offline behavioral analysis for xDSLs, we can extend the specification of the test oracles supported by the test runner.

Figure 7.5 illustrates this by showing the metamodel for the test cases comprising such extended test oracles. In fact, these oracles consist of multiple checkpoints to be validated during the test case, interleaved with the different event occurrences constituting the test scenario. These checkpoints can take one of the following form: (*i*) a temporal property, deployed as a runtime monitor and evaluated by the runtime monitoring backend, (*ii*) the reception from the model of a specific event occurrences or sequence thereof, or (*iii*) a comparison (using the trace comparison operator) between the trace of the tested model and a specified trace, both potentially modified through trace manipulation operators beforehand.

The addition of temporal properties allows to tie the emission of event occurrences to the dynamic state of the model. Including trace comparison after the potential filtering of dimensions and reduction of the traces enables precise non-regression testing.

CHAPTER 8

Conclusion and Perspectives

8.1 Conclusion

DSLs allow stakeholders to actively participate in the development of complex systems by providing them with the abstractions of their domain to model their part of the system. A large amount of DSLs are used to express the behavioral aspects of these systems. To be able to perform early V&V on such behavioral models, an execution semantics must be specified for the DSLs, resulting in languages called xDSLs. Executability alone is however not sufficient to enable V&V activities: an ecosystem of tools for behavioral analysis must also be provided to the modeler. However, xDSLs come in various shapes and forms, even more so than DSLs, due to their additional dimension corresponding to the specification of their execution semantics. This makes the provision of tools a tedious and error-prone task which must be repeated for each new xDSL, and therefore contributes to disincentivize the adoption of xDSLs. Therefore, a prerequisite to make the most out of xDSLs is an ecosystem of tools for behavioral analysis that can be reused across xDSLs.

We identified two main challenges to address to enable the provision of such an ecosystem of tools. First, to enable the provision of reusable tools for exploring the impact of the interaction that reactive models can have with their environment, an explicit and unified way to define this set of interaction is required at the language level (**Challenge#1**), *i.e.,* a metaprogramming approach for the definition of reactive DSLs. Second, to bolster the competitiveness of DSLs with regards to GPLs, providing appropriate tool support for the behavioral analysis of executable models that can be reused as-is or with little configuration across xDSLs is key **Challenge#2**.

Most approaches aiming to address similar challenges are either domain-specific, or limited in their scope of application, solving problems without providing reusable building blocks. In both cases, this hampers the provision of an ecosystem of tools, as in the former case, new tools must be developed for new xDSLs, and in the latter case, there is little potential interoperability between the tools. As a result, several new approaches leverage

or align with the recent advances in SLE that are LSP and DAP. From these approaches is emerging a model execution protocol that tends toward an adaptation of DAP to the field of MDE. These approaches aim to provide generic tooling for xDSLs that also integrates well with other potential tools built in a similar way, *i.e.,* they are built around an execution framework conforming to the aforementioned model execution protocol. Yet, the community effort supporting these approaches did not yet tackle a number of essential offline and online behavioral analysis features. Furthermore, these approaches seldom tackle the issue of behavioral analysis for reactive models. In light of this state of the art, we aimed in this thesis to tackle the considered challenges by providing an ecosystem of interoperable tools for the behavioral analysis of xDSLs, including reactive DSLs. To this end, we made the following three contributions.

First, we provided a unified way to define and expose the interactions that reactive models can have with their environment, to enable behavioral analysis of their reactive aspects. This is done under the form of a metalanguage for defining the behavioral interface implemented by an xDSL. To complement this, we define subtyping relationships between behavioral interfaces to enable the definition of event abstraction hierarchies bridging the abstraction gap between a DSL and a given behavioral interface. This contribution enables the definition of interaction-centric tools that are generic by reflection (through implementation relationships), and by abstraction (through subtyping relationships). Second, we propose four trace comprehension operators to help modelers analyze and understand the behavior of their reactive models from their execution trace. Two of these operators allow to manipulate execution traces to simplify them by filtering out noisy and redundant data. The other two are dedicated to the analysis of execution traces and help detect semantic variations between models or execution scenarios, as well as highlight cycles and bottleneck execution states. Third, we provide a backend for the runtime monitoring of xDSLs based on CEP and incremental model queries. We leverage this backend and take inspiration from the PSPs to provide a temporal property language that can be used by modelers to define temporal properties at the domain level. These properties are then translated to runtime monitors that can be deployed on the monitoring backend.

We integrated our three contributions in the GEMOC Studio to obtain an ecosystem of interoperable tools for the behavioral analysis of reactive DSLs. While the tools derived from our individual contributions are useful on their own, their combined use, and reuse as part of the extended test runner show the benefits of developing tools around a common protocol.

Overall, we addressed the two considered challenges and our contributions improve the state of the art regarding behavioral analysis facilities for reactive DSLs.

8.2 Perspectives

In the remainder of this chapter, we detail the envisioned perspectives for the work presented in this thesis, divided in two broad yet complementary categories: support for DSLs compiled to GPLs, and for DSLs for scientific computing.

8.2.1 Behavioral Analysis for Compiled DSLs

A number of DSLs are compiled to one or several target GPLs (*e.g.,* ThingML). While executable artifacts are produced from the models defined with such DSLs, these DSLs cannot actually be considered as executable. This is because the execution of artifacts produced from conforming models cannot be observed at the domain level.

As such, they cannot benefit from domain-level tools for behavioral analysis such as debugger, runtime monitors, or test runners. This in turn prevents domain experts to fully leverage the potential of DSLs unless they also have a technical background in programming with the target GPL. Even then, the discrepancy between the problem space and the solution space hampers the behavioral analysis of generated artifacts.

In [BW19], Bousse *et al.* address this in the case of DSLs compiled to another DSLs through a model transformation. In this case, the executable artifact produced from the source model is a model as well, meaning that it can be executed in accordance to the model execution protocol. This can be leveraged to realize an architecture where the execution of the target model provides feedback at the domain-level, which in turn allows to simulate the execution on the source model.

However, in the case of DSLs compiled to GPLs, designing a generic solution remains challenging. This is due to the fact that the generated artifacts must send feedback at the domain-level. This may require, for instance, a language-specific architecture to send this feedback, and to alter the code generator or connect and listen to the virtual machine in order to weave the corresponding code at build time or at runtime, respectively.

A solution to explore is a pivot feedback management architecture bridging the gap between DAP, which has already been implemented for several prevalent GPLs such as C/C++, C#, and Java, and the model execution protocol. This would enable to translate the target-level execution events received as part of DAP into domain-level execution events such as changes of the dynamic values of the model, or execution rules being called, thereby achieving domain-level feedback. The main benefit of such an approach is that it does not require to modify the code generator of the DSL. Instead, it requires to supply a translation scheme between the DAP execution events and the domain-level execution events.

8.2.2 Behavioral Analysis for Scientific Computing

To face the ongoing and upcoming environmental challenges resulting from climate change (*e.g.,* groundwater salinisation, drought, flooding), and to allow policy makers to make informed decisions about territorial developments, scientific modeling and computing are of the utmost importance. However, scientific models are mostly implemented in C++ or Fortran, a procedural language dedicated to scientific computing that does not provide any domain-level concept, the domains in questions being, for example, ecology, hydrology, meteorology or climatology.

For this reason, designing and debugging such models is a lengthy and error-prone process. Indeed, on one hand it requires both an expertise in C++/Fortran and in the scientific theories used to conceive the model, and on the other hand, even with such a

dual expertise, the mapping between the scientific domain (*i.e.,* the problem space) and the algorithmic domain (*i.e.,* the solution space) remains a complex one to carry out. In addition, running scientific simulations is a costly process that requires large amounts of energy, in a world where energy efficiency is of ever growing importance.

Introducing DSLs for scientific modeling is key in remedying this. Building on support for behavioral analysis for xDSLs compiled to languages such as Fortran or C++, such DSLs would only present domain abstractions and relieve modelers from algorithmic considerations when they implement their models. In turn, domain-level behavioral analysis enables the identification of incorrect behavior directly in the problem space, without requiring modelers to maintain a mental mapping between their algorithm and the model it realizes. Finally, the enhanced quality and efficiency of the generated simulation code would allow modelers to run their simulations with diminished energy costs.

APPENDIX A

Execution Semantics of Pattern/Scope Combinations

This appendix contains a compilation of the execution semantics for each temporal pattern and scope, given as quantified regular expressions and decision trees.

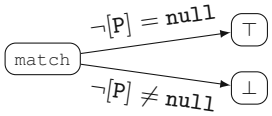
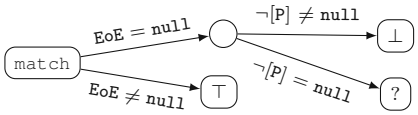
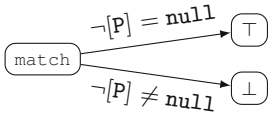
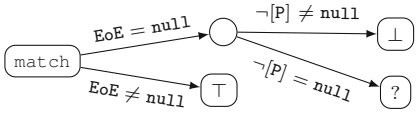
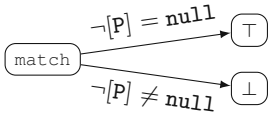
Pattern/scope combinations	QRE semantics and verdict procedure
always P **after** Q	EoE | (Q P* (¬[P] | EoE)) 
always P **after** Q **until** R	EoE | (Q P* (¬[P] | R | EoE)) 
always P **before** Q	EoE | Q | ¬[P] 
always P **between** Q **and** R	EoE | (Q P* ¬[P] P* (R | EoE) 
always P **globally**	EoE | ¬[P] 

Table A.1: Semantics of Universality patterns as QREs and verdict procedures.

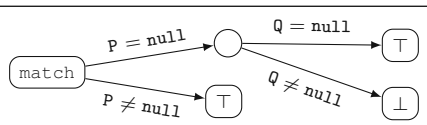
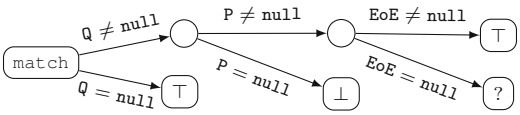
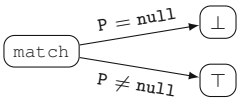
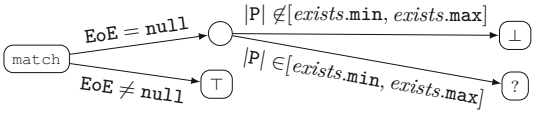
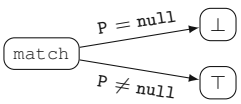
Pattern/scope combinations	QRE semantics and verdict procedure
exists [2,3] P **after** Q	EoE | (Q (¬[P]* P′ (¬[P]* P (¬[P]* P (¬[P]* P′ | EoE) | EoE) | EoE) | EoE)) 
exists [2,3] P **after** Q **until** R	EoE | (Q (¬[P]* P′ (¬[P]* P (¬[P]* P (¬[P]* P′ | R | EoE) | R | EoE) | R | EoE) | R | EoE)) 
exists [2,3] P **before** Q	¬[P]* P′ (¬[P]* P (¬[P]* P (¬[P]* P′ | Q | EoE) | Q | EoE) | Q | EoE) | Q | EoE 
exists [2,3] P **between** Q **and** R	EoE | Q ¬[P,R]* (P ¬[P,R]*)* (R | EoE) 
exists [2,3] P **globally**	¬[P]* P′ (¬[P]* P (¬[P]* P (¬[P]* P′ | EoE) | EoE) | EoE) | EoE 

Table A.2: Semantics of Existence patterns as QREs and verdict procedures.

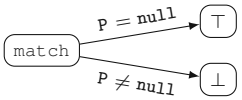
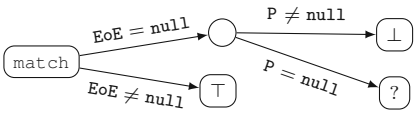
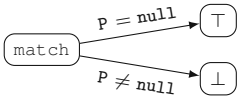
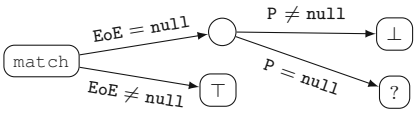
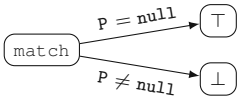
Pattern/scope combinations	QRE semantics and verdict procedure
never P **after** Q	EoE | (Q .*? (P | EoE)) 
never P **after** Q **until** R	EoE | (Q .*? (P | R | EoE)) 
never P **before** Q	EoE | Q | P 
never P **between** Q **and** R	EoE | (Q ¬[P]* (P ¬[P]*?)? (R | EoE) 
never P **globally**	EoE | P 

Table A.3: Semantics of Absence patterns as QREs and verdict procedures.

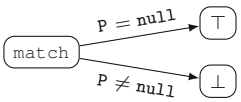
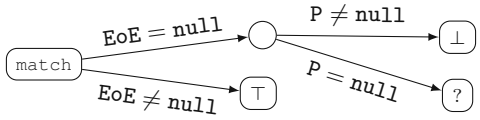
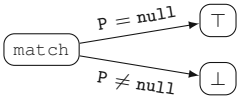
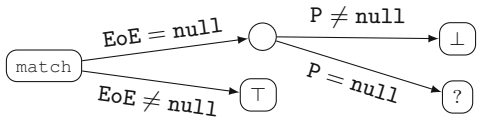
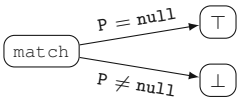
Pattern/scope combinations	QRE semantics and verdict procedure
S **precedes** P **after** Q	EoE | Q .*? (EoE | S | P) 
S **precedes** P **after** Q **until** R	EoE | Q .*? (EoE | R | S | P)) 
S **precedes** P **before** Q	EoE | Q | S | P 
S **precedes** P **between** Q **and** R	EoE | Q ¬[P,R,S]* (S ¬[R]* | P ¬[R]*)? (EoE | R) 
S **precedes** P **globally**	EoE | S | P 

Table A.4: Semantics of Precedence patterns as QREs and verdict procedures.

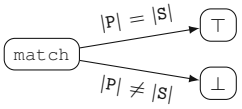
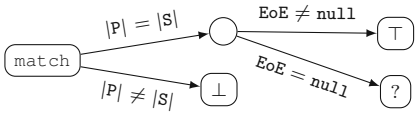
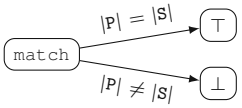
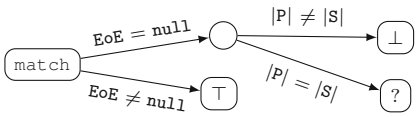
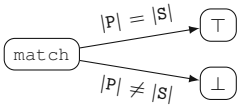
Pattern/scope combinations	QRE semantics and verdict procedure
S **respondsTo** P **after** Q	EoE | Q ¬[P]* (P ¬[S]* S ¬[P]*)* (P ¬[S]*)? EoE 
S **respondsTo** P **after** Q **until** R	EoE | Q ¬[P,R]* (P ¬[R,S]* S ¬[P,R]*)* (P ¬[R,S]*)? (R | EoE) 
S **respondsTo** P **before** Q	EoE | Q | (P ¬[S]* S ¬[P]*)* (P ¬[S]*)? (EoE | Q) 
S **respondsTo** P **between** Q **and** R	EoE | Q ¬[P,R]* (P ¬[R,S]* S ¬[P,R]*)* (P ¬[R,S]*)? (R | EoE) 
S **respondsTo** P **globally**	(P ¬[S]* S ¬[P]*)* (P ¬[S]*)? EoE 

Table A.5: Semantics of Response patterns as QREs and verdict procedures.

APPENDIX B

Runtime Monitoring Experimental Data

This appendix compiles the experimental data gathered from the monitoring of MiniJava, ThingML, and Activity Diagram models.

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
`Always P After Q`	2.1040	0.4121	0.6142	0.3426	0.4080	6.2868	0.8811
`Always P After Q Until R`	1.8162	0.4032	0.5235	0.2323	0.2202	9.4013	0.7544
`Always P Before Q`	1.1764	0.4015	0.3695	0.3105	0.3646	6.1629	0.7040
`Always P Between Q And R`	1.4978	0.2498	0.5272	0.3374	0.2439	9.0209	0.7260
`Always P Globally`	1.2148	0.3536	0.2381	0.2941	0.5437	3.1952	0.6114
`Exists P After Q`	1.2252	0.3432	0.3642	0.1129	0.3107	5.9110	0.5627
`Exists P After Q Until R`	1.4089	0.1889	0.5061	0.3065	0.5510	8.9489	0.7669
`Exists P Before Q`	1.4999	0.1955	0.5237	0.1227	0.2675	5.9338	0.5572
`Exists P Between Q And R`	1.4166	0.2428	0.5347	0.2944	0.3230	9.0156	0.7350
`Exists P Globally`	0.9930	0.0952	0.4273	0.0711	0.0805	3.0663	0.2986
`Never P After Q`	1.4442	0.2716	0.3216	0.2437	0.1155	6.0547	0.5273
`Never P After Q Until R`	1.3125	0.2028	0.3623	0.2986	0.1489	8.5939	0.5768
`Never P Before Q`	1.0856	0.2780	0.4328	0.1019	0.2936	6.0211	0.5353
`Never P Between Q And R`	1.3573	0.1632	0.5580	0.2770	0.3327	8.9424	0.6834
`Never P Globally`	1.0479	0.1833	0.1645	0.2005	0.2528	2.8345	0.4069
`S Precedes P After Q`	1.1304	0.2707	0.5999	0.2426	0.3397	8.6127	0.7120
`S Precedes P After Q Until R`	1.3375	0.4133	0.5990	0.3455	0.5997	12.3428	0.9727
`S Precedes P Before Q`	1.3462	0.2847	0.3433	0.2516	0.2674	8.9558	0.6555
`S Precedes P Between Q And R`	1.6022	0.3176	0.7122	0.3075	0.2060	12.3204	0.8102
`S Precedes P Globally`	1.1463	0.2255	0.3858	0.2075	0.0586	5.6620	0.4359
`S Responds To P After Q`	1.3272	0.1467	0.6743	0.2689	0.3438	9.0996	0.6927
`S Responds To P After Q Until R`	1.5661	0.3030	0.5126	0.1519	0.2339	12.5366	0.6904
`S Responds To P Before Q`	1.3059	0.1813	0.3199	0.0691	0.1447	8.6037	0.4321
`S Responds To P Between Q And R`	1.2707	0.3429	0.5676	0.1554	0.2016	12.2855	0.6758
`S Responds To P Globally`	1.4836	0.0368	0.6781	0.2453	0.2431	5.7224	0.4826

Table B.1: Relative execution overhead for MiniJava models (short executions).

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
Always P After Q	0.5441	0.7543	0.8169	0.4242	0.2603	1.2800	0.6016
Always P After Q Until R	0.4838	0.5379	0.5976	0.4614	0.4347	1.6665	0.6109
Always P Before Q	0.4354	0.7201	0.5471	0.4606	0.2438	1.2573	0.5379
Always P Between Q And R	0.3977	0.7794	0.8285	0.4530	0.3175	1.5289	0.6194
Always P Globally	0.1734	0.4535	0.4483	0.3800	0.3450	0.7893	0.3924
Exists P After Q	0.3421	0.6402	0.6937	0.3816	0.3244	1.0675	0.5213
Exists P After Q Until R	0.4098	0.6185	0.6007	0.3820	0.3880	1.4822	0.5676
Exists P Before Q	0.5663	0.6535	0.6941	0.3262	0.3390	0.9413	0.5468
Exists P Between Q And R	0.5621	0.7085	0.7754	0.3965	0.1734	1.4895	0.5623
Exists P Globally	0.3737	0.5572	0.6248	0.3300	0.1344	0.6812	0.3973
Never P After Q	0.4475	0.6036	0.6274	0.3300	0.1369	1.0247	0.4457
Never P After Q Until R	0.3567	0.4737	0.7030	0.1691	0.3979	1.2777	0.4658
Never P Before Q	0.2506	0.6502	0.7590	0.3583	0.3284	0.8229	0.4783
Never P Between Q And R	0.3883	0.6867	0.7874	0.3871	0.4001	1.2230	0.5842
Never P Globally	0.3723	0.5738	0.3467	0.2898	0.2700	0.6091	0.3902
S Precedes P After Q	0.3061	0.6329	0.6478	0.3642	0.3579	1.3748	0.5313
S Precedes P After Q Until R	0.4465	0.7335	0.8091	0.4135	0.3962	1.9015	0.6599
S Precedes P Before Q	0.4774	0.6796	0.6397	0.3666	0.2689	1.4164	0.5542
S Precedes P Between Q And R	0.3969	0.7798	0.5897	0.4025	0.2318	1.9074	0.5649
S Precedes P Globally	0.3175	0.6006	0.3907	0.3333	0.2810	1.0157	0.4383
S Responds To P After Q	0.4941	0.6767	0.7469	0.2044	0.3355	1.2634	0.5279
S Responds To P After Q Until R	0.4178	0.6812	0.7455	0.2243	0.4345	1.9014	0.5831
S Responds To P Before Q	0.3526	0.6733	0.5110	0.3797	0.3333	1.4018	0.5274
S Responds To P Between Q And R	0.4345	0.7356	0.7230	0.3903	0.4268	1.8303	0.6427
S Responds To P Globally	0.2447	0.3680	0.6235	0.2867	0.3091	0.9021	0.4062

Table B.2: Relative execution overhead for MiniJava models (long executions).

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
`Always P After Q`	0.8913	0.6932	0.8391	0.8342	0.8545	0.9443	0.8391
`Always P After Q Until R`	1.3005	0.6879	0.7159	0.9135	0.8984	0.8392	0.8725
`Always P Before Q`	0.8253	0.7415	0.7048	1.0323	0.9539	0.9019	0.8522
`Always P Between Q And R`	0.7008	0.7374	0.7438	0.8518	0.9537	0.9158	0.8117
`Always P Globally`	0.9282	0.5841	0.7304	0.8651	0.9686	1.0007	0.8321
`Exists P After Q`	0.9914	0.5022	0.7085	0.8638	0.8471	0.8263	0.7730
`Exists P After Q Until R`	0.9255	0.6593	0.5978	0.8854	0.8597	0.7541	0.7706
`Exists P Before Q`	0.9077	0.6051	0.5925	0.6815	0.8074	0.7484	0.7154
`Exists P Between Q And R`	0.7013	0.6912	0.7594	0.6793	0.7845	0.6673	0.7126
`Exists P Globally`	0.5722	0.4443	0.5961	0.7693	0.6996	0.7300	0.6249
`Never P After Q`	0.8930	0.6727	0.7352	0.9157	0.8466	0.7056	0.7892
`Never P After Q Until R`	0.7064	0.5771	0.6344	0.7402	0.7108	0.6294	0.6639
`Never P Before Q`	0.8038	0.4995	0.6293	0.7667	0.7435	0.8098	0.6990
`Never P Between Q And R`	0.5994	0.6282	0.5879	0.7536	0.7481	0.6330	0.6550
`Never P Globally`	0.6335	0.5539	0.5176	0.8440	0.6517	0.7225	0.6453
`S Precedes P After Q`	0.5634	0.7979	0.7386	0.7528	0.7738	0.8131	0.7347
`S Precedes P After Q Until R`	0.8111	0.7769	0.6130	0.7768	0.8866	0.7711	0.7680
`S Precedes P Before Q`	0.6424	0.4303	0.8812	0.7620	0.8219	0.7986	0.7041
`S Precedes P Between Q And R`	0.7434	0.6306	0.9330	0.8390	0.8903	0.7461	0.7904
`S Precedes P Globally`	0.7644	0.5061	0.7657	0.7637	0.7628	0.7338	0.7086
`S Responds To P After Q`	0.6949	0.6879	0.7387	0.8723	0.8635	0.9083	0.7892
`S Responds To P After Q Until R`	0.8148	0.7274	0.6990	0.8242	0.7893	0.8646	0.7845
`S Responds To P Before Q`	0.7845	0.6164	0.6755	0.8415	0.7620	0.7810	0.7395
`S Responds To P Between Q And R`	0.7615	0.6472	0.7213	0.8927	0.8517	0.7873	0.7727
`S Responds To P Globally`	0.6589	0.7513	0.6502	0.8765	0.8710	0.9389	0.7831

Table B.3: Relative execution overhead for ThingML models (short executions).

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
`Always P After Q`	0.6656	0.8010	0.7640	0.6665	0.6464	0.6393	0.6945
`Always P After Q Until R`	0.5652	0.7266	0.7580	0.5871	0.5661	0.6134	0.6316
`Always P Before Q`	0.6649	0.8427	0.6260	0.6270	0.6005	0.6129	0.6577
`Always P Between Q And R`	0.6600	0.7921	0.7733	0.5810	0.6445	0.6250	0.6750
`Always P Globally`	0.6690	0.6348	0.6399	0.5624	0.6181	0.6013	0.6200
`Exists P After Q`	0.6306	0.6873	0.6625	0.6329	0.6410	0.6037	0.6425
`Exists P After Q Until R`	0.5031	0.7279	0.6372	0.5944	0.5762	0.5187	0.5883
`Exists P Before Q`	0.5846	0.5780	0.6377	0.5902	0.6382	0.5320	0.5923
`Exists P Between Q And R`	0.5729	0.6519	0.7294	0.5719	0.5895	0.5213	0.6026
`Exists P Globally`	0.5918	0.6367	0.5152	0.4279	0.5036	0.5017	0.5252
`Never P After Q`	0.6016	0.7364	0.6784	0.5956	0.5614	0.5955	0.6254
`Never P After Q Until R`	0.5897	0.6997	0.6381	0.6019	0.4845	0.5124	0.5832
`Never P Before Q`	0.5577	0.6886	0.6140	0.5506	0.4777	0.5104	0.5624
`Never P Between Q And R`	0.5021	0.6627	0.5789	0.5934	0.4808	0.5174	0.5525
`Never P Globally`	0.4791	0.5713	0.6065	0.5073	0.4613	0.4947	0.5176
`S Precedes P After Q`	0.5249	0.6731	0.7268	0.5108	0.5559	0.5884	0.5917
`S Precedes P After Q Until R`	0.5598	0.7539	0.6876	0.6653	0.5878	0.5558	0.6309
`S Precedes P Before Q`	0.6202	0.6768	0.7195	0.5228	0.4970	0.5273	0.5881
`S Precedes P Between Q And R`	0.6205	0.7332	0.6917	0.6840	0.5867	0.5511	0.6413
`S Precedes P Globally`	0.4640	0.6059	0.5690	0.5666	0.4816	0.5082	0.5301
`S Responds To P After Q`	0.6406	0.7134	0.8062	0.6817	0.5975	0.6346	0.6758
`S Responds To P After Q Until R`	0.6779	0.7468	0.6692	0.6682	0.5867	0.5589	0.6483
`S Responds To P Before Q`	0.6614	0.6236	0.6873	0.6166	0.5851	0.6307	0.6333
`S Responds To P Between Q And R`	0.6745	0.6180	0.6696	0.6704	0.5862	0.5632	0.6287
`S Responds To P Globally`	0.6401	0.7078	0.7072	0.6253	0.5811	0.6240	0.6460

Table B.4: Relative execution overhead for ThingML models (long executions).

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
Always P After Q	2.2160	1.2421	2.5808	2.1821	2.8521	109.1307	4.1106
Always P After Q Until R	2.4373	1.1938	2.6407	2.0916	2.7643	161.9735	4.3939
Always P Before Q	1.6510	1.3127	2.1276	1.8635	2.7113	108.8471	3.6928
Always P Between Q And R	1.6001	1.5924	2.7264	2.1007	2.9357	157.2368	4.3458
Always P Globally	1.2322	0.8977	1.8281	1.4916	1.9716	54.1924	2.6184
Exists P After Q	1.4086	1.3149	1.8934	2.1865	2.1846	105.1669	3.4753
Exists P After Q Until R	1.5173	1.2046	2.2886	2.1457	2.5043	162.0876	3.9227
Exists P Before Q	2.3385	1.3374	2.2906	1.8333	2.5571	103.8437	3.8942
Exists P Between Q And R	2.2047	1.6471	2.7270	1.7445	2.2658	171.6824	4.3441
Exists P Globally	2.1090	0.9641	2.0690	1.7129	1.8500	55.2448	3.0051
Never P After Q	2.0819	1.3181	1.9148	1.8340	2.2217	101.7618	3.6005
Never P After Q Until R	1.6479	1.2122	2.1499	1.9434	2.1490	162.7454	3.7804
Never P Before Q	1.9042	1.2411	2.1282	1.9193	2.2086	107.5073	3.6311
Never P Between Q And R	2.2014	1.1741	2.1936	1.9275	2.4337	157.6227	4.0155
Never P Globally	1.2725	0.7960	1.2364	1.4999	1.5591	51.5195	2.3074
S Precedes P After Q	1.4715	1.4612	2.4775	1.5929	2.4587	167.3053	3.8948
S Precedes P After Q Until R	1.8954	1.2260	2.8087	2.0446	2.7081	220.2100	4.4682
S Precedes P Before Q	1.1174	1.0570	2.0873	1.7695	2.0521	156.5591	3.3453
S Precedes P Between Q And R	1.6999	1.5607	3.5175	2.2733	2.6411	222.3964	4.8150
S Precedes P Globally	1.2264	0.8772	1.9734	1.3925	2.0849	103.4663	2.9339
S Responds To P After Q	1.5804	1.4638	2.7959	2.1944	2.8692	163.2464	4.3362
S Responds To P After Q Until R	1.9944	1.0181	3.5985	1.9451	2.6077	221.0622	4.4899
S Responds To P Before Q	1.6215	1.2742	2.4284	3.0056	2.3260	162.5864	4.2268
S Responds To P Between Q And R	1.6075	1.2107	2.8480	2.6915	2.6084	222.2841	4.5307
S Responds To P Globally	1.1899	1.0052	1.7248	1.9154	1.8902	113.5205	3.0765

Table B.5: Relative execution overhead for Activity Diagram models (short executions).

Model size	S	M		L			mean
Property footprint	S	S	M	S	M	L	
Always P After Q	0.4748	0.8953	1.0425	1.6157	1.8788	17.9889	1.7007
Always P After Q Until R	0.4933	1.0271	1.2364	1.5728	2.0658	25.3726	1.9297
Always P Before Q	0.4270	0.9684	1.2786	1.5924	2.0337	17.9798	1.7703
Always P Between Q And R	0.4578	1.0078	1.2539	1.6506	2.0032	24.9410	1.9044
Always P Globally	0.3415	0.6378	0.9416	1.2541	1.4994	10.4073	1.2606
Exists P After Q	0.4814	0.8382	1.0531	1.6581	1.6470	14.7204	1.6048
Exists P After Q Until R	0.4384	0.7584	1.1929	1.5594	1.6463	22.5846	1.6863
Exists P Before Q	0.3730	0.8217	1.1058	1.6136	1.6552	15.2146	1.5482
Exists P Between Q And R	0.4538	0.9201	1.1635	1.5165	1.7635	22.5061	1.7552
Exists P Globally	0.3451	0.8175	0.7077	1.4537	1.2197	7.5368	1.1777
Never P After Q	0.3143	0.7586	0.8680	1.4380	1.6085	15.1472	1.3912
Never P After Q Until R	0.4412	0.9444	1.0700	1.9574	1.5571	23.1877	1.7773
Never P Before Q	0.2790	0.8732	0.7793	1.6043	1.5106	15.0664	1.3809
Never P Between Q And R	0.4633	0.8220	1.0174	1.5118	1.7649	22.2991	1.6870
Never P Globally	0.2314	0.5682	0.7076	1.1126	1.2817	7.8859	1.0076
S Precedes P After Q	0.4544	0.7490	1.0503	1.3888	1.6100	22.7690	1.6219
S Precedes P After Q Until R	0.3928	0.9342	1.2612	1.5236	1.7977	30.4979	1.8388
S Precedes P Before Q	0.5059	0.8113	0.8740	1.5415	1.5839	22.7976	1.6471
S Precedes P Between Q And R	0.5257	0.9278	1.2271	1.5754	1.8910	30.2018	1.9433
S Precedes P Globally	0.2340	0.6795	0.9253	1.2144	1.3025	14.6628	1.2270
S Responds To P After Q	0.3734	0.7587	1.2219	1.4987	1.6820	22.8402	1.6466
S Responds To P After Q Until R	0.4440	0.9909	1.2410	1.6961	1.8102	29.9980	1.9213
S Responds To P Before Q	0.4450	0.7840	1.0877	1.5595	1.6486	22.5281	1.6736
S Responds To P Between Q And R	0.4686	0.9788	1.2887	1.5635	1.6800	30.7041	1.9042
S Responds To P Globally	0.4141	0.7462	1.0418	1.2874	1.3028	14.6325	1.4113

Table B.6: Relative execution overhead for Activity Diagram models (long executions).

List of Figures

List of Tables

List of Algorithms

Listings

Bibliography

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010.

[ACC14] Mathieu Acher, Benoit Combemale, and Philippe Collet. Metamorphic domain-specific languages: A journey into the shapes of a language. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 243–253. ACM, 2014.

[AK03] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[AMP18] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Inferring hierarchical motifs from execution traces. 2018.

[BAE+15] Ahmed Barnawi, Ahmed Awad, Amal Elgammal, Radwa El Shawi, Abduallah Almalaise, and Sherif Sakr. BP-MaaS: A Runtime Compliance-Monitoring System for Business Processes. In *BPM (Demos)*, 2015.

[BBG+60] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[BCBB15] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: modeling and generating model slicers. *Software & Systems Modeling*, 14(1):321–337, 2015.

[BCC+15] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proceedings of the International Conference on Software Language Engineering (SLE'15)*. ACM, 2015.

[BCCG07] Réda Bendraou, Benoit Combemale, Xavier Crégut, and Marie Pierre Gervais. Definition of an executable SPEM 2.0. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 390–397. IEEE, 2007.

[BDV+16] 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, page 8, Amsterdam, Netherlands, October 2016.

[Béz04] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[Béz05] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[BGPS12] Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *ICSE'12*. IEEE, 2012.

[BKD09] Johannes Bohnet, Martin Koeleman, and Jürgen Döllner. Visualizing massively pruned execution traces to facilitate trace exploration. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 57–64. IEEE, 2009.

[BLC+18] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. Omniscient debugging for executable dsls. *Journal of Systems and Software*, 137:261 – 288, 2018.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[BMCB17] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. Advanced and efficient execution trace management for executable domain-specific modeling languages. *Software & Systems Modeling*, pages 1–37, 2017.

[BS02] Claus Brabrand and Michael I Schwartzbach. *Growing languages with metamorphic syntax macros*, volume 37. ACM, 2002.

[BSE10] Nils Bandener, Christian Soltenborn, and Gregor Engels. Extending dmm behavior specifications for visual execution and debugging. In *International Conference on Software Language Engineering*, pages 357–376. Springer, 2010.

[BSVV18] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries for runtime monitoring of cyber-physical systems. In *FASE'18*. Springer, Cham, 2018.

[BSVV19] Márton Búr, Gábor Szilágyi, András Vörös, and Dániel Varró. Distributed graph queries over models@ run. time for runtime monitoring of cyber-physical systems. *STTT*, 2019.

[Bün19] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *Proceedings of the 7\(^{th}\) International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 131–142, 2019.

[BURV11] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A Graph Query Language for EMF Models. In *Theory and Practice of Model Transformations - 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, 2011.

[BvdA12] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. Process diagnostics using trace alignment: Opportunities, issues, and challenges. *Inf. Syst.*, 37(2):117–141, 2012.

[BW19] Erwan Bousse and Manuel Wimmer. Domain-level observation and control for compiled executable dsls. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 150–160. IEEE, 2019.

[CDB+14] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. A solution to the ttc'15 model execution case using the gemoc studio. In *8th Transformation Tool Contest*. CEUR, 2014.

[CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[CFAI17] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. *arXiv preprint arXiv:1708.07229*, 2017.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. Viatra-visual automated transformations for formal verification and validation of uml models. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 267–270. IEEE, 2002.

[CJGK+18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[CLCM00] Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *ACM Sigplan Notices*, volume 35, pages 130–145. ACM, 2000.

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):1–62, 2012.

[Com15] Benoit Combemale. *Towards Language-Oriented Modeling*. PhD thesis, 2015.

[CSW08] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: a foundation for language driven development., 2008.

[CTI15] Valerio Cosentino, Massimo Tisi, and Javier Luis Cánovas Izquierdo. A model-driven approach to generate external dsls from object-oriented apis. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 423–435. Springer, 2015.

[CZvD10] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, 2010.

[DAC98] 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*. ACM, 1998.

[DAH01] Luca De Alfaro and Thomas A Henzinger. Interface automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001.

[DCB+17] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures (COMLAN)*, 49:176 – 195, 2017.

[Dea16] Julien Deantoni. Modeling the behavioral semantics of heterogeneous languages and their coordination. In *2016 Architecture-Centric Virtual Integration (ACVI)*, pages 12–18. IEEE, 2016.

[DH04] James B Dabney and Thomas L Harman. *Mastering simulink*. Pearson, 2004.

[DK98] Arie Van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[DK07] Dolev Dotan and Andrei Kirshin. Debugging and testing behavioral uml models. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 838–839. ACM, 2007.

[dNVN+12] Leandro Marques do Nascimento, Daniel Leite Viana, PAS Neto, Dhiego AO Martins, Vinicius Cardoso Garcia, and Silvio RL Meira. A systematic mapping study on domain-specific languages. In *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA'12)*, pages 179–187, 2012.

[DREP12] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Model transformations. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 91–136. Springer, 2012.

[DT16] Zoé Drey and Ciprian Teodorov. Object-oriented design pattern for DSL program monitoring. In *SLE'16*. ACM, 2016.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, volume 46, pages 391–406. ACM, 2011.

[EVDSV+15] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[FGLP10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical language analysis in software linguistics. In *International Conference on Software Language Engineering*, pages 316–326. Springer, 2010.

[Flo93] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop Theory and Application of Graph Transformations (TAGT'98)*, volume 1764, pages 157–167, 2000.

[Fow05] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[GCD+12] Clément Guy, Benoît Combemale, Steven Derrien, Jim R. H. Steel, and Jean-Marc Jézéquel. On model subtyping. In *8th European Conference on Modelling Foundations and Applications (ECMFA)*, pages 400–415, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: a framework for the development of textual domain specific languages. In *Companion of the 30th international conference on Software engineering*, pages 925–926. ACM, 2008.

[GPL+06] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java concurrency in practice*. Pearson Education, 2006.

[GV90] Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *International Colloquium on Automata, Languages, and Programming*, pages 626–638. Springer, 1990.

[H+96] Paul Hudak et al. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[HBRV10] Abel Hegedus, Gábor Bergmann, István Ráth, and Dániel Varró. Back-annotation of simulation traces with change-driven model transformations. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 145–155. IEEE, 2010.

[HLL06] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190. IEEE, 2006.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnuelli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and Mark Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[HRV12] Ábel Hegedüs, István Ráth, and Dániel Varró. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica Electrical Engineering and Computer Science*, 56(3):71–82, 2012.

[J+75] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[JCB+13] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software & Systems Modeling (SoSyM)*, 14(2), 2013.

[JEA+07] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 2007.

[KDRPP09] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models (CVSM'09)*, pages 1–6, 2009.

[Kle08] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[KPP08] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer, 2008.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular development of textual domain specific languages. In *International Conference on Objects, Components, Models and Patterns*, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International journal on software tools for technology transfer*, 12(5):353–372, 2010.

[KSB08] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–173. Springer, 2008.

[KV10] Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *ACM sigplan notices*, 45(10):444–463, 2010.

[KVDSV09] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009.

[LBM+18] Dorian Leroy, Erwan Bousse, Anaël Megna, Benoit Combemale, and Manuel Wimmer. Trace comprehension operators for executable dsls. In *European Conference on Modelling Foundations and Applications*, pages 293–310. Springer, 2018.

[LBW+20] Dorian Leroy, Erwan Bousse, Manuel Wimmer, Tanja Mayerhofer, Benoit Combemale, and Wieland Schwinger. Behavioral interfaces for executable dsls. *Software and Systems Modeling*, 2020.

[LDCM15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (bcool). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195. IEEE, 2015.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[LJB+20] Dorian Leroy, Pierre Jeanjean, Erwan Bousse, Manuel Wimmer, and Benoit Combemale. Runtime monitoring for executable dsls. *Journal of Object Technology*, 19(2), 2020.

[LJH06] Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *Australian Software Engineering Conference (ASWEC'06)*, pages 10–pp. IEEE, 2006.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, page 1, 2001.

[LMK14] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. Semantic model differencing utilizing behavioral semantics specifications. In *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems (MODELS'14)*, pages 116–132, 2014.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5), 2009.

[Luc02] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[McC93] John McCarthy. Towards a mathematical science of computation. In *Program Verification*, pages 35–56. Springer, 1993.

[MDDV16] Bart Meyers, Joachim Denil, István Dávid, and Hans Vangheluwe. Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2016.

[MDL+14] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Promobox: A framework for generating domain-specific property languages. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 1–20, 2014.

[Mer13] Marjan Mernik. An object-oriented approach to language compositions for software language engineering. *Journal of Systems and Software*, 86(9):2451–2464, 2013.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering (SLE)'13*, volume 8225 of *LNCS*. Springer, 2013.

[Mos01] Peter D Mosses. The varieties of programming language semantics and their uses. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 165–190. Springer, 2001.

[MRR11] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Addiff: semantic differencing for activity diagrams. In *Proceedings of the SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*, pages 179–189, 2011.

[MVG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[NCM03] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. Polyglot: An extensible compiler framework for java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007.

[Obj13a] Object Management Group. OMG Unified Modeling Language (OMG UML), V 2.5, September 2013. `http://www.omg.org/spec/UML/2.5`.

[Obj13b] Object Management Group. Semantics of a Foundational Subset for Executable UML Models, V 1.1, August 2013.

[Obj14] Object Management Group (OMG). Object Constraint Language (OCL), Version 2.4, February 2014. `http://www.omg.org/spec/OCL/2.4`.

[Obj16] Object Management Group. Meta Object Facility (MOF) Core Specification, V 2.5, June 2016. `http://www.omg.org/spec/MOF/2.5`.

[Obj19] Object Management Group. Precise Semantics of UML State Machines Specification, V 1.0, May 2019.

[Omg08] QVT Omg. Meta object facility (MOF) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[OMG13] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, 2013.

[Par13] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[PB02] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380. ACM, 2018.

[RG09] Lukas Renggli and Tudor Gîrba. Why smalltalk wins the host languages shootout. In *Proceedings of the International Workshop on Smalltalk Technologies*, pages 107–113. ACM, 2009.

[Rob01] Eric Roberts. An overview of minijava. *ACM SIGCSE Bulletin*, 33(1), 2001.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Eclipse Series. Addison-Wesley Professional, 2008.

[Sch87] David A Schmidt. *Denotational Semantics: A Methodology for Language Development by Phil*. McGraw-Hill Professional, 1987.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[SE09] Michael Soden and Hajo Eichler. Towards a model execution framework for eclipse. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, page 4. ACM, 2009.

[SGC+09] Jocelyn Simmonds, Yuan Gan, Marsha Chechik, Shiva Nejati, Bill O'Farrell, Elena Litani, and Julie Waterhouse. Runtime monitoring of web service conversations. *IEEE Transactions on Services Computing*, 2(3), 2009.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling (SoSym)*, 6(4):401–413, Dec 2007.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.

[TCKI99] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. Openjava: A class-based macro system for java. In *Workshop on Reflection and Software Engineering*, pages 117–133. Springer, 1999.

[VBD+13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[VC15] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1–40, 2015.

[vdA11] Wil M. P. van der Aalst. *Basics of Applied Stochastic Processes*. Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, 2011.

[VDA12] Wil Van Der Aalst. Process mining: making knowledge discovery process centric. *ACM SIGKDD Explorations Newsletter*, 13(2):45–49, 2012.

[vdBvDH+01] Mark GJ van den Brand, Arie van Deursen, Jan Heering, HA De Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A Olivier, Jeroen Scheerder, et al. The asf+ sdf meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, 2001.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[VMP14] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[Voe14] Markus Voelter. *Generic tools, specific languages*. Citeseer, 2014.

[VWT+14] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabrieël Konat. A language designer's workbench: a one-stop-shop for implementation and verification of language designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 95–111. ACM, 2014.

[War94] Martin P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[WKV14] Guido H Wachsmuth, Gabriël DP Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE software*, 31(5):35–43, 2014.

[ZLG05] Jing Zhang, Yuehua Lin, and Jeff Gray. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*, pages 199–217. Springer, 2005.